Operating System/2™ Programming Tools and Information Version 1.2

Procedures Language/2 REXX Programming Reference

64F0291

Operating System/2™
Programming Tools
and Information
Version 1.2

Procedures Language/2
REXX Programming Reference

First Edition (1989)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Armonk NY 10504.

Operating System/2 and OS/2 are trademarks of the International Business Machines Corporation.

© Copyright International Business Machines Corporation 1986, 1987, 1989
All rights reserved.

About This Book

The Operating System/2 Version 1.2 (OS/2) *Procedures Language 2/REXX Programming Reference* is a technical reference for professional programmers who develop system programs and applications using calls for the Restructured EXtended eXecutor (REXX) language.

This book also provides some guidance information on how to use the calls. It is intended to be used with the Operating System/2 Version 1.2 *Programming Guide*.

Prerequisite Knowledge

This book is intended for application developers who know how to program in REXX. The information also assumes you are new to programming with OS/2 and the Presentation Manager. You should understand the OS/2 services available to users. Further information on these can be obtained from the OS/2 Product library.

Conventions used in Call Descriptions

The documentation of each call contains these sections:

Call name The call name at the top of each page, followed by a brief description of the call.

Parameters Each parameter is listed with its data type and a brief description.

There are three kinds of parameters:

Input Specified by the programmer.

Output Returned by OS/2.

Input/Output Specified by the programmer and modified by OS/2.

A list of possible return codes or errors (where appropriate) is included in this section. Some functions do not have return codes or error messages.

Remarks Additional information about the function, where required.

Bindings C and Assembler languages.

Example An example is shown in C language for some functions.

Programming Note: The functions in this book are named in mixed-case for readability, but are known to the system as uppercase character strings. If you are using a compiler that generates a mixed-case external name, you should code the OS/2 function calls in uppercase.

Sample Programs

The OS/2 tools package contains sample programs and files that demonstrate common programming techniques using REXX. The programs are:

CALLREXX.CMD Calls REXX from a C program
RXMACDLL.CMD Uses the external functions interface
RXSUBDLL.CMD Uses the subcommand interface.

Another file of interest is RXSTRING.DOC, which explains the RXSTRING.LIB library.

Related Publications

Three books that you will find useful along with this book are:

- The Operating System/2 Version 1.2 *Procedures Language 2/REXX User's Guide*, available separately.
- The Operating System/2 Version 1.2 *Procedures Language 2/REXX Reference*, available separately.
- *The REXX Language: A Practical Approach to Programming*, by M. F. Cowlishaw, published by Prentice-Hall, Inc.

The Operating System/2 Version 1.2 *Programming Overview* introduces the programming concepts that you should understand before you begin developing applications to run on OS/2, and describes the set of books, tools, programming aids, and sample programs that make up the OS/2 Programming Tools and Information Library.

Details of publications for OS/2 Version 1.2 and related products are shown in the Library Diagram on page v.

OS/2 Product

IBM Operating System/2
Standard Edition
Version 1.2

Getting Started
Using Advanced Features
Product Information

6024926 3.5″ diskette
6024930 5.25″ diskette

OS/2 Programming Tools and Information

IBM Operating System/2
Version 1.2

Installation
Programming Overview
Programming Guide
Building Programs
Dialog Tag Language Guide and Reference
Dialog Manager Guide and Reference
Dialog Manager and Dialog Tag Language Reference Summary
Programming Reference (3 books)
Bindings Reference for Presentation Manager (4 books for COBOL/2, FORTRAN/2, C/2, and Macro Assembler/2)
I/O Subsystems and Device Support (2 books)
Systems Application Architecture Common User Access: Advanced Interface Design Guide

6024929

Separate Order (no charge)

Keyboards and Code Pages

6280345

Available Separately

Command Reference

6024928

Service Coordinator's Guide

15F2214

Programming Languages

IBM Basic Compiler/2
Version 1.0 6280179

IBM Macro Assembler/2
Version 1.0 6280181

IBM Pascal Compiler/2
Version 1.0 6280183

IBM FORTRAN/2
Version 1.02 6280185

IBM COBOL/2
Version 1.0 6280207

IBM C/2
Version 1.1 6280284

Systems Application Architecture

An Overview

GC26-4341

Writing Applications: A Design Guide

SC26-4362

Common User Access: Advanced Interface Design Guide
SC26-4582

Common User Access: Basic Interface Design Guide
SC26-4583

Common Programming Interface

C Reference - Level 2

SC09-1308

COBOL Reference

SC26-4354

Dialog Reference

SC26-4356

FORTRAN Reference

SC26-4357

Procedures Language Reference
SC26-4358

Presentation Reference

SC26-4359

Notes:

1. *CodeView™ is available with IBM C/2 Version 1.1 and upon request. An order form is available with the OS/2 Standard Edition Version 1.2 Product. It is also compatible with the Macro Assembler/2, Pascal Compiler/2, and BASIC Compiler/2 listed above.*

™ Trademark of Microsoft Corporation

2. *IBM C/2 Version 1.1 is available on 5.25 in. diskette. An order form is available in the C/2 package.*
3. *See your IBM representative for ordering information.*

Contents

Chapter 1. Introduction to the REXX API 1-1
Using the REXX APIs 1-1
Subcommands 1-1
Functions 1-1
System Exits 1-1
Macrospace Interface 1-1
Variable Pool Interface 1-2
General Characteristics 1-2
Data Types and Structures 1-2

Chapter 2. Invoking the REXX Interpreter 2-1
From OS/2 2-1
From Within an Application 2-1
REXXSAA – Call Interpreter 2-2

Chapter 3. Subcommand Interface 3-1
Writing Subcommand Handlers 3-1
Invocation 3-1
Duplicate Environment Names 3-2
Search Order 3-2
Subcommand Interface Calls 3-2
Sample Subcommand Calls 3-3
RxSubcomDrop – Deregister Subcommand Environment 3-4
RxSubcomExecute – Send Commands to Subcommand Environment 3-5
RxSubcomLoad – Load Subcommand Environment 3-7
RxSubcomQuery – Query Subcommand Environment Name 3-8
RxSubcomRegister – Register Subcommand Environment 3-10

Chapter 4. External Functions 4-1
Registering Function Packages 4-1
Dynamic Link Library Functions 4-1
Executable Functions In Memory 4-1
Writing External Functions 4-2
Returning Result Values 4-2
Calling External Functions 4-2
External Interface Calls 4-3
Sample External Function Calls 4-3
RxFunctionCall – Call External Function 4-4
RxFunctionDeregister – Remove Function 4-6
RxFunctionQuery – Query List of Functions 4-7
RxFunctionRegister – Register Function 4-8

Chapter 5. Macrospace Interface 5-1
Search Order 5-1
Pre-External Function Registration 5-1
Post-External Function Registration 5-1
Storage of Macrospace Libraries 5-1
Macrospace Interface Calls 5-2
RxMacroChange – Change or Add Function to Macrospace 5-3
RxMacroDrop – Delete Function from Macrospace 5-4
RxMacroErase – Erase Functions in Macrospace 5-5
RxMacroLoad – Load Functions into Macrospace 5-6
RxMacroQuery – Search for Function in Macrospace 5-8
RxMacroReOrder – Change Search-Order in Macrospace 5-9
RxMacroSave – Save Functions in Macrospace 5-10

Chapter 6. Variable Pool Interface 6-1
RxVar – Pass Request Blocks to Variable Pool 6-3
Interface Types 6-4

Symbolic Interface 6-4
Direct Interface 6-4
Shared Variable Functions 6-4
Setting Variables 6-4
Fetching Variables 6-4
Dropping Variables 6-5
Setting an Exit Return Value 6-6
Notes and Limits 6-6

Chapter 7. System Exits 7-1
Writing System Exit Handlers 7-1
System Exit Interface Calls 7-2
Exits 7-2
Exit Characteristics 7-4
Exit Definitions 7-4
RXFNC – Process External Functions 7-5
RXFNCCAL — Process external functions 7-5
RXCMD – Process Host Commands 7-7
RXCMDHST — Process host commands 7-7
RXMSQ – Manipulate Queue 7-9
RXMSQPLL — Pull a line from the queue 7-9
RXMSQPSH — Place a line on the queue 7-9
RXMSQSIZ — Return the number of lines in the queue 7-10
RXMSQNAM — Set the name of the active queue 7-11
RXSIO – Session I/O 7-12
RXSIOSAY — Output result of SAY to standard output stream 7-12
RXSIOTRC — TRACE output processing/call to output TRACE results 7-12
RXSIOTRD — Read from STDIN stream 7-13
RXSIODTR — Debug read 7-14
RXHLT – Halt Processing 7-15
RXHLTCLR — Clear halt indicator 7-15
RXHLTTST — Test halt indicator 7-15
RXTRC – Test External Trace Indicator 7-17
RXTRCTST — Test external trace indicator 7-17
RXINI – Initialization Processing 7-18
RXINIEXT — Initialization processing 7-18
RXTER – Termination Processing 7-19
RXTEREXT — Termination processing 7-19
Asynchronous Request Interface 7-19
RxHaltSet – Inform Halt Condition 7-20
RxTraceSet – Turn On Debug Mode 7-21
RxTraceReset – Turn Off Interactive Debug Mode 7-22

Appendix. Errors Returned from REXX Calls A-1

Index X-1

Chapter 1. Introduction to the REXX API

This book describes the Procedures Language/2 REXX Application Programming Interface (API) for OS/2 and discusses the following topics:

- Invoking the REXX Interpreter
- Subcommand processing
- External functions
- Macrospace interface
- Variable pool interface
- System exits.

Note: Throughout this book the term *application* is used to refer to programs written in languages other than REXX. Examples of calling conventions are given in the C and Assembler programming languages. The functions described are, however, also available to PASCAL.

Using the REXX APIs

The features described in this book allow an application to extend many parts of the REXX language. This includes providing handlers for subcommand, external function, and system exit processing. These handlers are similar in the ways that they must be coded, compiled, and packaged.

In addition, REXX provides a macrospace interface that allows applications to store and execute REXX routines in memory and thereby minimize disk access. Also, REXX provides the means for applications to manipulate the variables in REXX programs through the variable pool interface.

Subcommands

Subcommands are single clauses consisting of an expression. The expression is evaluated and the result is passed as a command string to the currently addressed environment. Subcommands are used in the context of REXX running as a macro processor under some environment or program. Refer to Chapter 3, "Subcommand Interface" on page 3-1 for more information.

Functions

Functions are direct extensions to the capabilities of REXX. An application (including REXX itself) can provide a function to augment the function set of REXX. Unlike subcommands, which generally correspond to the application's normal command set, functions are generally specifically for REXX and may only have a meaning from within REXX. They may be as simple as an encapsulation of several base REXX instructions or as complex as a large library of functions outside REXX and written in some other programming language. Refer to Chapter 4, "External Functions" on page 4-1 for more information.

System Exits

System exits allow REXX to operate under programmer-defined variations of the operating system. By using these exits, specific REXX actions can be executed as calls to routines provided by the caller of the interpreter, instead of using code of the interpreter itself. Refer to Chapter 7, "System Exits" on page 7-1 for more information.

Macrospace Interface

There are times when REXX is the language best suited for implementing an external function, such as when the main purpose of a function is to manipulate strings. Normally, REXX external functions are stored on a system's hard disk. If the function is called repeatedly, the hard-disk search and load time can add significant overhead to the total execution time of the calling REXX procedure.

The macrospace interface lets an application store REXX procedures in memory so they are executed

there, thereby avoiding hard-disk overhead. These external functions can be thought of as high-usage macros; they can be loaded into a macro space. Refer to Chapter 5, "Macrospace Interface" on page 5-1 for more information.

Variable Pool Interface

REXX provides an API to allow certain applications, such as subcommands and external functions, to read and modify the current set of variables being used by a REXX procedure. Variables can be read, modified, and dropped. All variables can be queried. Certain private REXX information, such as the version or source strings, can be queried. The applications provide control information to the provided interface that instruct REXX what changes to make. REXX will then process the request internally; the applications do not need to know the contents of the REXX variable pool to make direct changes. All changes are made indirectly through the shared variable request block. Refer to Chapter 6, "Variable Pool Interface" on page 6-1 for more information.

General Characteristics

Here are some basic requirements for applications to be used as handlers for subcommand, external function, and system exit processing:

- A handler must be written as a large model program. In addition, it must:
  - Use its caller's stack segment
  - Set up its own data segment
  - Be entered by a far call
  - Obey the PASCAL calling conventions.

  Note: Under the IBM C/2 Compiler Version 1.1 the -Alfu compiler switches force the registers to be properly set up.
- A handler must be *packaged*, either as:
  - Part of an OS/2 Dynamic Link Library (a *dynalink* or DLL), or
  - Part of an executable (.EXE) module.
- A handler must be *registered* with REXX before it can be used. The registration tells REXX where the handler is and how it can be invoked. For example, the information provided on the registration of a dynalink external function includes the name of the function, the name of the dynalink library containing the handler, and the name of the procedure within the dynalink that implements the function. Also note:
  - Dynalink handlers are universal to the OS/2 system; they can be invoked from a REXX procedure running in any process in any OS/2 session.
  - .EXE handlers are local to the registering process; this means that handlers packaged as part of an .EXE module can only be invoked by a REXX procedure running in the same process that registered it.

Data Types and Structures

Many of the calls described in this book require a method by which arguments may be passed. The data structure named **RXSTRING** is used for communication between the REXX interpreter and application programs. The arguments passed may consist of short strings (strings residing entirely within one segment) or huge strings (single strings spanning multiple segments). The following structure and macros are used to describe an **RXSTRING**.

C Language

```
/***    Structure for external interface string (RXSTRING) */


#define RXAUTOBUFLEN         256L

#define RXNULLSTRING(r)      (!(r).strptr)
#define RXZEROLENSTRING(r)   ((r).strptr && !(r).strlength)
#define RXVALIDSTRING(r)     ((r).strptr && (r).strlength)
#define RXSTRLEN(r)          (RXNULLSTRING(r)?0L:(r).strlength)
#define RXSTRPTR(r)          (r).strptr
#define MAKERXSTRING(r,p,l)  (r).strptr=(PCH)p;(r).strlength=(ULONG)l


typedef struct {
   ULONG           strlength;          /*   length of string        */
   PCH             strptr;             /*   far pointer to string   */
   } RXSTRING;

typedef RXSTRING FAR *PRXSTRING;       /* pointer to a RXSTRING      */
```

Assembler Language

```
;***    Structure for external interface string (RXSTRING)
;
RXAUTOBUFLEN        equ 00000100h
;
RXSTRING struc
strlength           dd  ?               ; length of string
strptr              dd  ?               ; far pointer to string
RXSTRING ends
;
```

Notes:

1. REXX arguments either may be given a value (which may be null, '') or not specified.
   - If an argument is specified, the appropriate parts of the **RXSTRING** will be given values.
   - If a null character ('') is specified as the argument, *strlength* will be zero, and *strptr* will be non-zero.
   - If no argument is specified, *strptr* will be zero.
2. The huge mode of **RXSTRING** contains the same information as the short mode, except that the memory is allocated through a call to DosAllocHuge. Through this convention, strings in a user's program can span multiple segments.
3. In certain instances (such as subcommand calls, function calls, and some exit calls), the interpreter will provide an **RXSTRING** with a *strlength* and buffer size of 250 for the return of information. If the data for return from these calls is less than 250 characters in length, the program should copy the data into this buffer and reset the *strlength* field accordingly.

Chapter 2. Invoking the REXX Interpreter

The REXX interpreter can be called directly from the operating system or from within an application.

From OS/2

The standard OS/2 CMD.EXE shell calls the interpreter for the user:

- At OS/2 command prompts.
- In calls from CMD (batch) files. Use the OS/2 **CALL** command to invoke a REXX program in a batch file if you want control to return to the caller.
- From the Desktop Manager window in Presentation Manager.

From Within an Application

REXXSAA is the function that calls the REXX interpreter from an application program. Procedures run in the same session, process, and thread as the caller.

The REXX processor is an OS/2 Dynamic link library (DLL) that is fully re-entrant and allows recursive entry by the same thread in a process. The processor supports different procedures running independently in multiple threads within the same process.

A C-language prototype for calling REXX is included in the file REXXSAA.H in the installation package. Likewise, an Assembler-language prototype is included in the file REXXSAA.INC.

REXXSAA –
Call Interpreter

This call allows the user to call the REXX interpreter from an application program.

REXXSAA (ArgCount, ArgList, ProgramName, Instore, EnvName, CallType, Exits, ReturnCode, Result)

Parameters

ArgCount *(SHORT)* – input
The number of arguments supplied for this call to the REXX interpreter.

ArgList *(PRXSTRING)* – input
An array of **RXSTRING** items that describes the arguments for this call to the REXX interpreter.

ProgramName *(PSZ)* – input
A pointer to an ASCIIZ string containing at least the filename and, optionally, the extension and full drive and path specification of the REXX procedure. If no file extension is specified, a default of ".CMD" is supplied. If no path is given, the normal OS/2 file search (current directory then environment path) will be performed. For example: C:\UTIL\PLAYTIME.CMD.

Instore *(PRXSTRING)* – input
An array of two descriptors for in-storage REXX procedures:

Instore[0] An **RXSTRING** describing the memory buffer containing the source of the procedure. The source must be in the same format as if it were on disk (that is, complete with carriage returns, line feeds, and end-of-file characters [implied at end-of-string]).

Instore[1] A second **RXSTRING**, used initially to return the tokenized image of the source in DosAllocSeg or DosAllocHuge space from the interpreter to the caller. Subsequently, the caller can pass this tokenized image back to the interpreter. The interpreter will use this tokenized image and thus save the pre-tokenization time. See Remarks below.

If *Instore* is specified, no disk search will be performed; *ProgramName* will be used only as the name of the procedure. If no instorage buffer is to be supplied, the *Instore pointer* should be null.

EnvName *(PSZ)* – input
A pointer to a string specifying the initial interpreter address environment. If no environment name is specified, the initial address will default to the file extension. If specified, this value must be in the form of an ASCIIZ string.

CallType *(SHORT)* – input
A flag describing how REXX was called; possible values are:

RXCOMMAND	as a command
RXSUBROUTINE	as a subroutine
RXFUNCTION	as a function.

Exits *(PRXSYSEXIT)* – input
An array of **RXSYSEXIT** elements that contains pointers to subcommand environment names and an integer code identifying the exit. The last **RXSYSEXIT** element must contain the code **RXENDLST** that indicates the end of the system exit list. Set this parameter to null if there are no system exits provided. See Chapter 7, "System Exits" on page 7-1 for system exit details.

ReturnCode *(PSHORT)* – input
If the *Result* string is numeric, it will be converted to an integer and also returned in *ReturnCode*.

Result *(PRXSTRING)* – output
The **RETURN** or **EXIT** value from the *ProgramName* program.

Returns

An integer return code from a call to REXXSAA may be issued by the REXX interpreter or from OS/2. The possible return codes are:

negative	Interpreter errors. These are the standard SAA-defined interpreter error codes. Refer to Appendix, Errors Returned from REXX Calls for a description of these error conditions.
0	No errors occurred. Procedure executed normally.
positive	OS/2 return code indicating problems finding or loading the interpreter. See the return codes for the OS/2 functions DosLoadModule and DosGetProcAddr for details.

Remarks

With regard to in-storage procedures:

- Memory allocated by *Instore* must be released through DosFreeSeg by the caller; once the tokenized image is returned to the caller the memory is no longer tracked by REXX.
- Whenever the source is changed, the tokenized image must be released and *Instore*[*1*] must specify an empty **RXSTRING**.
- If *Instore*[*1*] is specified, *Instore*[*0*] need not actually contain the source (that is, it may be empty).
- To prevent in-storage tokenization, set *Instore*[*1*] to an invalid **RXSTRING**.

C Language

```
short     argc, rc;
char      *rexxsaafile, *envnam;
RXSTRING  *argv, instor, retval;
RXSYSEXIT *sysexit;

return_code = REXXSAA(argc, argv, rexxsaafile, instor, envnam, RXCOMMAND, sysexit, &rc, &retstr);
```

Assembler Language

```
@REXXSAA          macro   ac,av,rf,ins,env,ct,sex,rc,ret
                  @define  REXXSAA
                  @pushw   ac        ; Num of args passed to rexx
                  @pushs   av        ; Array of args passed to rexx
                  @pushs   rf        ; [d:][path] filename[.ext]
                  @pushs   ins       ; Loc of rexx proc in memory
                  @pushs   env       ; ASCIIZ initial environment.
                  @pushw   ct        ; type (command,subrtn,funct)
                  @pushs   sex       ; SysExit env. names &  codes
                  @pushs   rc        ; Ret code from if numeric
                  @pushs   ret       ; Retvalue from the rexx proc.
                  call    far ptr REXXSAA
                 endm
;
```

Chapter 3. Subcommand Interface

An application can be dynamically made known to the REXX interpreter by name as an environment accessible by the REXX ADDRESS instruction. To do this, the application must be registered as a specific subcommand environment. This can be accomplished through the RXSUBCOM interface.

Writing Subcommand Handlers

Because subcommands are closely associated with a given application, they are written as part of the application. The subcommand code can reside in the same module (.EXE or DLL) as the application, or it can reside in an accessory dynalink. Any separate procedure that can be registered with REXX and follows the appropriate conventions can be a handler.

For details on the requirements and options for packaging API handlers, see "General Characteristics" on page 1-2. The *library name* must be the same as the file name of the dynalink or .EXE files being used.

Documentation of subcommand handlers packaged as dynalink libraries must specify the *library name* (used by DosLoadModule) of the dynalink created; see note 1 on page 3-2. This *library name* must be used in order to register the subcommand and may be needed in order to resolve duplicate environment names. The procedure name of the subcommand handler must also be published. This procedure name must be exported from the dynalink library.

Invocation

Here is a sample subcommand handler definition and invocation:.

```
SHORT APIENTRY my_subcom (
        RXSTRING,       /* Command string passed from the caller    */
        PUSHORT,        /* pointer to short for return of flags     */
        PRXSTRING       /* pointer to RXSTRING for return string    */
        );

unsigned short pascal far my_subcom (command, flags, retstr)
RXSTRING command;
unsigned short far *flags;
RXSTRING far *retstr;
```

Where:

command is the entire command string as resolved by REXX.

flags are flags returned by the subcommand handler to the interpreter to indicate successful completion, error, or failure conditions.

RXSUBCOM_OK	Subcommand completed normally
RXSUBCOM_ERROR	Error in subcommand
RXSUBCOM_FAILURE	Failure in subcommand

retstr the address of a **RXSTRING** to define your result in. See "Returning Result Values" on page 4-2 for additional information.

These arguments correspond to the same arguments defined for the function RxSubcomExecute, described on page 3-5.

- The subcommand handler is passed the command as a **RXSTRING**, the pointer to the flag storage area and the pointer to a **RXSTRING** for the subcommand result string.
- The subcommand returns an integer return code that is placed in the REXX special variable RC or returned to the caller of RxSubcomExecute in the last (RC) argument.
  - If a non-zero number is returned, that number is used as the return code.

- If zero is returned and the result **RXSTRING** is not empty, the return code is set to the value of the result **RXSTRING**.
- Otherwise the return code is set to zero.

Duplicate Environment Names

Multiple subcommand handlers can register themselves for any given subcommand environment name. However, for any given OS/2 "boot" there are unique *library names* associated with individual handlers packaged in DLLs (see 1). These *library names* should be documented and published with the instructions for the subcommand handlers. To make subcommand environment names unique the user can incorporate this documented *library name* into the subcommand environment addressed.

For example, assume that the fictitious QED Corporation produces an editor whose subcommand environment is EDIT, packages it in QEDIT.DLL, and gives it a library name of QEDIT. At the same time, the fictitious XYZ Corporation produces a word processor whose subcommand environment is also EDIT, but packages it as XWORD.DLL with a library name of XWORD.

The subcommand handler to be used can be identified by providing the library name when addressing the subcommand environment. For example, the REXX subcommand `ADDRESS 'Edit.Qedit'` will send subcommands to the EDIT environment established by the QEDIT dynalink. Likewise, the subcommand `ADDRESS 'Edit.Xword'` will send subcommands to the EDIT environment established by the XWORD dynalink.

Notes:

1. The *library names* mentioned here are the ModuleNames used on DosLoadModule calls. These names are found along the OS/2 LIBPATH path so there can be only one handler of *library name* found until LIBPATH is changed and the system re-booted or libraries are unloaded and moved around in the subdirectories searched through LIBPATH.
2. Providing library qualifying information is only necessary for duplicate dynalink subcommand environments because of the established subcommand environment search order.

Search Order

Because multiple subcommand handlers can register by the same environment name, it is necessary to establish a subcommand handler search order.

When a dynalink library name is specified with the ADDRESS instruction, REXX invokes the handler registered for that environment and packaged in that library. If the specified handler is not available, an error is returned.

If no dynalink library name is specified:

- The registered handler chain is searched for a matching environment name that was registered from the current process ID and session ID (essentially an .EXE subcommand handler entry point).
- If there is no local handler, search the chain for the first matching handler that can be found and used.
- If none is found, return an error.

Subcommand Interface Calls

The calls that you use for registering and manipulating subcommand interfaces are:

- RxSubcomDrop (Deregister Subcommand Environment)
- RxSubcomExecute (Send Commands to Subcommand Environment)
- RxSubcomLoad (Load Subcommand Environment)
- RxSubcomQuery (Query Subcommand Environment Name)
- RxSubcomRegister (Register Subcommand Environment).

Sample Subcommand Calls

```
char far *name, far *dllname;
unsigned short exist, flags, rc, return_code;
double   userword;
SCBLOCK  scb;
RXSTRING command, retstr;


return_code = RxSubcomRegister (&scb);
return_code = RxSubcomDrop     (name, dllname);
return_code = RxSubcomLoad     (name, dllname);
return_code = RxSubcomQuery    (name, dllname, &exist, &userword);
return_code = RxSubcomExecute  (name, dllname, command, &flags, &rc, &retstr);
```

The calls are described on the remaining pages of this chapter.

RxSubcomDrop – Deregister Subcommand Environment

This call allows the user to deregister a subcommand environment.

RxSubcomDrop (EnvName, ModuleName)

Parameters

EnvName *(PSZ)* – input
A pointer to a null terminated (ASCIIZ) subcommand environment name string.

ModuleName *(PSZ)* – input
A pointer to a null terminated (ASCIIZ) string containing the library name of the dynalink library containing the registered subcommand handler. This name may be a NULL if there are no duplicate subcommand environment names registered. If there are multiple subcommand handlers for a given environment name and a library name is not provided, the search order documented in "Search Order" on page 3-2 is used for environment resolution. Also see "Duplicate Environment Names" on page 3-2.

Returns

0	RXSUBCOM_OK
30	RXSUBCOM_NOTREG
40	RXSUBCOM_NOCANDROP
1003	RXSUBCOM_BADTYPE

Remarks

The environment is dropped or deregistered from the active subcommand environment list.

C Language

```
/***    RxSubcomDrop - Drop a Subcommand environment */

USHORT APIENTRY RxSubcomDrop(
        PSZ,                        /* Name of the Environment    */
        PSZ);                       /* DLL Module Name            */
```

Assembler Language

```
;*  RxSubcomDrop - Drop a Subcommand environment
;
;*  @RxSubcomDrop  *Environment, *Dllname
;
@RxSubcomDrop       macro   env,dll
                    @define RXSUBCOMDROP
                    @pushs  env         ; Name of the Environment
                    @pushs  dll         ; DLL Module Name
                    call    far ptr RXSUBCOMDROP
                    endm
;
```

RxSubcomExecute – Send Commands to Subcommand Environment

This call allows the user to send commands to a subcommand environment.

RxSubcomExecute (EnvName, ModuleName, Command, Flags, RetVal, Result)

Parameters

EnvName *(PSZ)* – input
A pointer to a null terminated (ASCIIZ) subcommand environment name string.

ModuleName *(PSZ)* – input
A pointer to a null terminated (ASCIIZ) string containing the library name of the dynalink library containing the registered subcommand handler. This name may be a NULL if there are no duplicate subcommand environment names registered. If there are multiple subcommand handlers for a given environment name and a library name is not provided, the search order documented in "Search Order" on page 3-2 is used for environment resolution. Also see "Duplicate Environment Names" on page 3-2.

Command *(RXSTRING)* – input
A string describing the entire command string.

Flags *(PUSHORT)* – output
A pointer to an integer of flags, which is used to notify the caller of failure to locate a subcommand or of errors encountered while executing the subcommand. This information is reflected by the interpreter during **TRACE** (errors) activity. See RxSubcom Flag Definitions on page 3-1 for definitions of the flags.

RetVal *(PUSHORT)* – output
A pointer to the return code from the subcommand handler.

Result *(PRXSTRING)* – output
A pointer to the result of the subcommand processing. The subcommand handler modifies the *Result* structure for the returned string.

Note: The string returned from the subcommand handler is allocated segments which are given to the caller. The caller uses DosFreeSeg to free the returned string when finished with it. See "Returning Result Values" on page 4-2.

Returns

0	RXSUBCOM_OK
30	RXSUBCOM_NOTREG
50	RXSUBCOM_LOADERR
127	RXSUBCOM_NOPROC

Remarks

The REXX interpreter invokes the appropriate subcommand handler for the specified subcommand environment name and library (if needed).

RxSubcomExecute – Send Commands to Subcommand Environment

C Language

```
/***    RxSubcomExecute - Execute a command in an environment */

USHORT APIENTRY RxSubcomExecute(
        PSZ,                        /* Name of Subcommand Environ */
        PSZ,                        /* Module name of its' DLL    */
        PRXSTRING,                  /* Command string to be passed*/
        PUSHORT,                    /* Stor for error flag notice */
        PUSHORT,                    /* Stor for rc from handler   */
        PRXSTRING );                /* Stor for returned string   */

#endif /* INCL_RXSUBCOM */
```

Assembler Language

```
;*  RxSubcomExecute - Execute a command in an environment
;
;*  @RxSubcomExecute  *Environment, *Dllname, *Cmdstring, *flag,
;                     *rc, *return
;
@RxSubcomExecute    macro   env,dll,cmd,flag,rc,ret
                    @define RXSUBCOMEXECUTE
                    @pushs  env         ; Name of Subcommand Environ
                    @pushs  dll         ; Module name of its' DLL
                    @pushs  cmd         ; Command string to be passed
                    @pushs  flag        ; Stor for error flag notice
                    @pushs  rc          ; Stor for rc from handler
                    @pushs  ret         ; Stor for returned string
                    call    far ptr RXSUBCOMEXECUTE
                    endm
;
```

RxSubcomLoad – Load Subcommand Environment

This call allows the user to load a subcommand environment packaged in a dynalink.

RxSubcomLoad (EnvName, ModuleName)

Parameters

EnvName *(PSZ)* – input
A pointer to a null terminated (ASCIIZ) subcommand environment name string.

ModuleName *(PSZ)* – input
A pointer to a null terminated (ASCIIZ) string containing the library name of the dynalink library containing the registered subcommand handler. This name may be a NULL if there are no duplicate subcommand environment names registered. If there are multiple subcommand handlers for a given environment name and a library name is not provided, the search order documented in "Search Order" on page 3-2 is used for environment resolution. Also see "Duplicate Environment Names" on page 3-2.

Returns

0	RXSUBCOM_OK
30	RXSUBCOM_NOTREG
50	RXSUBCOM_LOADERR
127	RXSUBCOM_NOPROC

Remarks

If a subcommand environment is registered by library and routine name, a caller may wish to have the REXX interpreter issue an explicit load for the environment-handling routines.

C Language

```
/***    RxSubcomLoad - Load a Subcommand environment */

USHORT APIENTRY RxSubcomLoad(
        PSZ,                        /* Name of the Environment    */
        PSZ);                       /* DLL Module Name            */
```

Assembler Language

```
;*  RxSubcomLoad - Load a Subcommand environment
;
;*  @RxSubcomLoad  *Environment, *Dllname
;
@RxSubcomLoad       macro   env,dll
                    @define RXSUBCOMLOAD
                    @pushs  env         ; Name of the Environment
                    @pushs  dll         ; DLL Module Name
                    call    far ptr RXSUBCOMLOAD
                    endm
;
```

RxSubcomQuery — Query Subcommand Environment Name

This call allows the user to query a subcommand environment name.

RxSubcomQuery (EnvName, ModuleName, Flag, UserWord)

Parameters

EnvName *(PSZ)* — input
A pointer to a null-terminated (ASCIIZ) subcommand environment name string.

ModuleName *(PSZ)* — input
A pointer to a null-terminated (ASCIIZ) string containing the library name of the dynalink library containing the registered subcommand handler. This name may be a NULL if there are no duplicate subcommand environment names registered. If there are multiple subcommand handlers for a given environment name and a library name is not provided, the search order documented in "Search Order" on page 3-2 is used for environment resolution. Also see "Duplicate Environment Names" on page 3-2.

Flag *(PUSHORT)* — output
A pointer to an indicator of whether the subcommand identified by *EnvName* is registered or not. If *Flag* is set to 0, the subcommand is not registered. If *Flag* is set to 1, the subcommand is registered.

UserWord *(double far *)* — output
The double word of data saved as *scbuser* during the registration process.

Returns

0	RXSUBCOM_OK
30	RXSUBCOM_NOTREG
1003	RXSUBCOM_BADTYPE

Remarks

The caller may Query the status of a particular environment name. The caller passes a pointer to the name of an environment. A flag indicating the existence of the subcommand (1 = registered, 0 = not registered) is returned as well as the userword saved at registration.

C Language

```
/***    RxSubcomQuery - Query an environment for Existence */

USHORT APIENTRY RxSubcomQuery(
        PSZ,                        /* Name of the Environment    */
        PSZ,                        /* DLL Module Name            */
        PUSHORT,                    /* Stor for existence code    */
        double far *);              /* Stor for user word         */
```

Assembler Language

```
;*  RxSubcomQuery - Query an environment for Existence
;
;*  @RxSubcomQuery  *Environment, *Dllname, *Existence, *User
;
@RxSubcomQuery      macro   env,dll,ex,usr
                    @define RXSUBCOMQUERY
                    @pushs  env         ; Name of the Environment
                    @pushs  dll         ; DLL Module Name
                    @pushs  ex          ; Stor for existence code
                    @pushs  usr         ; Stor for user word
                    call    far ptr RXSUBCOMQUERY
                    endm
;
```

RxSubcomRegister – Register Subcommand Environment

This call allows the user to register a subcommand environment.

RxSubcomRegister (SubComBlock)

Parameters

SubComBlock *(PSCBLOCK)* – input
A pointer to a user-allocated **SCBLOCK** (see -- Heading 'SCBDEFS' unknown --).

Data Definitions

To register a subcommand environment, a subcommand request block or **SCBLOCK** must be created. Here is the format of an **SCBLOCK**:

C Language

```
/***    Structure of REXX Subcommand Block (SCBLOCK) */

typedef struct subcom_node {
    struct subcom_node far *next;      /* Pointer to the next block   */
    PSZ    scbname;                    /* Subcom environment name     */
    PSZ    scbdll_name;                /* Subcom module name          */
    PSZ    scbdll_proc;                /* Subcom procedure name       */
    double scbuser;                    /* User area                   */
    PFN    scbaddr;                    /* Subcom environment address  */
    USHORT scbmod_handle;              /* Dynalink module handle      */
    USHORT scbdrop_auth;               /* Permission to drop          */
    PID    scbpid;                     /* Process ID of registrant    */
    USHORT scbsid;                     /* Session ID.                 */
    } SCBLOCK;

typedef SCBLOCK FAR *PSCBLOCK;
```

Assembler Language

```
;***    Structure of Rexx Subcommand Block (SCBLOCK)
;
SCBLOCK struc
next                dd  ?              ; pointer to the next block
scbname             dd  ?              ; subcom environment name
scbdll_name         dd  ?              ; subcom module name
scbdll_proc         dd  ?              ; subcom procedure name
scbuser             dq  ?              ; user area
scbaddr             dd  ?              ; subcom environment address
scbmod_handle       dw  ?              ; dynalink module handle
scbdrop_auth        dw  ?              ; Permission to drop
scbpid              dw  ?              ; Pid of Registrant
scbsid              dw  ?              ; Session ID.
SCBLOCK ends
;
PSCBLOCK            dd  ?              ; Pointer to an SCBLOCK
;
```

Within the **SCBLOCK** you must specify either:

- **For dynalink registration:** the subcommand environment name (*scbname*), the library name (*scbdll_name*), and the procedure name (*scbdll_proc*); or
- **For EXE-module registration:** the subcommand environment name (*scbname*), and the handler address (*scbaddr*).

Where:

scbname An ASCIIZ string to be registered as the subcommand environment name.

scbdll_name An ASCIIZ string specifying the library name of the dynalink library containing the subcommand handler. *scbdll_name* is the field that REXX uses to determine if this is a DLL or .EXE registration. If *scbdll_name* is a null pointer (0), this is assumed to be an .EXE registration. If *scbdll_name* is non-null, this is assumed to be a DLL registration. The library name is the name supplied by the module definition file for this library.

scbdll_proc An ASCIIZ string specifying the procedure name of the subcommand handler. Used by DosGetProcAddr to get the address of the subcommand handler procedure.

scbaddr The actual address of the subcommand environment handler. This parameter is for use by .EXE programs supplying an internal routine as the subcommand handler. These handlers are "local" to the current process (that is, only subcommands originating in the same process as the registration process of "local" handlers will be presented to the handler).

The following fields are optional:

scbdrop_auth Drop authority. Determines which processes can drop the subcommand handler. The options are:

RXSUBCOM_DROPPABLE
Allows any process to drop this subcommand handler using the RxSubcomDrop call (see page 3-4). This is the default setting.

RXSUBCOM_NONDROP
Only a thread of execution with the same session ID and process ID as the thread that registered this handler will be allowed to complete a RxSubcomDrop request against this handler.

scbuser User area to be saved with subcommand registration information.

Returns

0	RXSUBCOM_OK
10	RXSUBCOM_DUP
1002	RXSUBCOM_NOEMEM
1003	RXSUBCOM_BADTYPE

C Language

```
/***   RxSubcomRegister - Register environment as Subcommand handler */

USHORT APIENTRY RxSubcomRegister(
        PSCBLOCK );                  /* Pointer to SCBLOCK         */
```

Assembler Language

```
;*  RxSubcomRegister - Register environment as Subcommand handler
;
;*  @RxSubcomRegister  *SubcomBlock
;
@RxSubcomRegister   macro   pscb
                    @define RXSUBCOMREGISTER
                    @pushs  pscb       ; Pointer to SCBLOCK
                    call    far ptr RXSUBCOMREGISTER
                    endm
;
```

Chapter 4. External Functions

There are two types of external functions, routines written in REXX and routines written in other OS/2-supported languages. External functions written in REXX are not registered with REXX; they are found by a disk search for a file name that matches the function name being called. Functions written in other languages, however, must be registered. Whether they are packaged as part of an .EXE module or as a procedure within a dynalink library, REXX must be told where they are so that the correct invocation mechanism can be used.

Note: Both kinds of external functions are subject to a standard search order for all subroutines and functions.

Registering Function Packages

The parameters needed for each kind of registration are described here. The actual interfaces for registration and execution are described on page 4-3.

For details on the general requirements and options for packaging API handlers, see "General Characteristics" on page 1-2.

Dynamic Link Library Functions

A dynamic link library file has a name and one or more externally callable entry points. The name and entry points are defined both in the function source code and in the .DEF file provided to the linker.

Functions available in DLLs are registered by:

REXX-visible name — an ASCIIZ-string name which will be called within a REXX program; for example,

```
Say MyBeep(200,200)
```

Dynamic Link Library — an ASCIIZ-string name which identifies the DLL file containing the requested function; for example,

```
MYDLL
```

The DLL file must exist in a directory included in the LIBPATH system-environment variable, which is set from CONFIG.SYS when the system is booted.

Entry point — an ASCIIZ-string name which identifies the visible or available entry point into the DLL used to access the proper function; for example,

```
USHORT APIENTRY MYENTRYPOINT(...)
```

If multiple functions are to be accessed through the same entry point, the proper function name can be determined from the first parameter of RxFunctionCall.

Executable Functions In Memory

Functions that are part of executable files loaded in memory are registered by:

REXX-Visible Name — an ASCIIZ-string name which will be called within a REXX program; for example,

```
Say MyBeep(200,200)
```

Transfer address — the address of a function in memory which will accept the proper parameters and perform the required operations for the function call; for example,

```
(PSZ)(&MyFunction())
```

Writing External Functions

Here is the format for writing an external function:

```
SHORT APIENTRY my_rexx_function (PSZ, SHORT, PRXSTRING, PSZ, PRXSTRING);

short far pascal my_rexx_function (name, argc, argv, queue, retstr);
char         * name;            /* name the function is invoked as     */
short          argc;            /* number of arguments                 */
RXSTRING far   argv[];          /* argument array                      */
char     far * queue;           /* name of current queue               */
RXSTRING far * retstr;          /* RXSTRING for function result        */
```

Where

name is the name by which the function is invoked.
argc is the size of the argument list, namely the number of elements passed.
argv is an array of RXSTRINGs describing the arguments.
queue is the name of the queue to be used by the function.
retstr is the result RXSTRING; see "Returning Result Values."

Functions must return a valid RXSTRING for the *retstr* parameter. If there is no value to return from the function, an empty RXSTRING can be used. If called as a function and no value is returned, the interpreter will generate error 44, "Function did not return data."

For an RXSTRING in huge form, the storage area must be created by a call to DosAllocHuge(). For RXSTRING in short form, the storage area must be created by a call to DosAllocSeg(). The interpreter will destroy any RXSTRING passed to it as a return value from a function call. Any program that wishes to maintain the information passed to REXX must create its own copy before transferring control back to the interpreter.

Returning Result Values

Subcommands and external functions are capable of returning **RXSTRING** result strings which are returned in the special variable **RC** for subcommands or in **RESULT** for external subroutines (external function results will be processed in line to the calling clause). To accomplish this, function and subcommand handlers will be passed an address to an **RXSTRING** of length 250 in the REXX interpreter's storage space. The processor can use this **RXSTRING** or return a far pointer to memory allocated by DosAllocSeg or DosAllocHuge. If this memory is allocated by a process other than the one in which the interpreter is running (for example, during execution of another program called by the function), it must be made available to the interpreter via a call to DosGiveSeg. The interpreter frees this storage after copying the result string into the appropriate interpreter storage.

The possible **RXSTRING** result values are:

- A null *strptr* (that is, *strptr* = zero) is assumed to represent no result returned.
- A *strlength* of zero with a valid *strptr* is assumed to represent a zero-length result returned.
- A non-zero *strlength* with a valid *strptr* is assumed to represent a valid result returned.

Calling External Functions

Functions registered with REXX by address can only be used in a program running in the same process, that is, one having the same process identification. It is possible to have different functions registered with the same REXX-visible name, as long as they are all registered by address and are all registered from different processes. If, however, a function is registered as part of a dynamic link library, it is a global function, and its REXX-visible name cannot be repeated.

External Interface Calls

The calls that you use for registering and manipulating function packages are:

- RxFunctionCall (Call Function)
- RxFunctionDeregister (Remove Function)
- RxFunctionQuery (Query List of Functions)
- RxFunctionRegister (Register Function).

Sample External Function Calls

```
#define NULL_PTR 0L
char     *func, *dll, *proc, *queue;
ushort   argc;
short    rc, return_code;
RXSTRING *argv, retstr;

return_code = RxFunctionRegister   (func, dll, proc, RX_DYNALINK);
return_code = RxFunctionRegister   (func, NULL_PTR, (PSZ) &my_rexx_function, RX_CALLENTRY);
return_code = RxFunctionDeregister (func);
return_code = RxFunctionQuery      (func);
return_code = RxFunctionCall       (func, argc, argv, queue, &rc, &retstr);
```

The calls are described on the remaining pages of this chapter.

RxFunctionCall – Call External Function

This call allows the user to call a function registered with REXX as an external function.

RxFunctionCall (FuncName, ArgCount, ArgList, RetVal, Result, QueueName)

Parameters

FuncName *(PSZ)* – input
A pointer to the ASCIIZ string containing the name of the external function to call.

ArgCount *(USHORT)* – input
The number of arguments being passed to the function.

ArgList *(PRXSTRING)* – input
A pointer to the list of arguments being passed to the function.

RetVal *(PUSHORT)* – output
A pointer to the address of a place for the function to put a return code.

Result *(PRXSTRING)* – output
A pointer to the descriptor for the result string from the function (See "Returning Result Values" on page 4-2).

QueueName *(PSZ)* – input
A pointer to the default queue for use by the function.

Returns

0	RXFUNC_OK
30	RXFUNC_NOTREG
40	RXFUNC_MODNOTFND
50	RXFUNC_ENTNOTFND

Remarks

The function invoked will be the most recent registration of *FuncName*, with certain restrictions. If a function is found with a registered name of *FuncName*, but is not available to the current process, the search for the function will continue as if no registration had been found.

A function registered with the **RXFUNC_DYNALINK** flag will be executed if it is the first valid registration found.

C Language

```
/***    RxFunctionCall - Call a function in the AFT */

USHORT APIENTRY RxFunctionCall (
        PSZ,                        /* Name of function to call   */
        USHORT,                     /* Number of arguments        */
        PRXSTRING,                  /* Array of argument strings  */
        PUSHORT,                    /* RC from function called    */
        PRXSTRING,                  /* Storage for returned data  */
        PSZ );                      /* Name of active data queue  */


#endif /* INCL_RXFUNC */
```

Assembler Language

```
;*  RxFunctionCall - Call a function in the AFT
;
;*  @RxFunctionCall *Name,Argc,*Argv,*rc,*ret,*Qname
;
@RxFunctionCall     macro name,argc,argv,rc,ret,qname
                    @define RXFUNCTIONCALL
                    @pushs  name        ; Name of function to call
                    @pushw  argc        ; Number of arguments
                    @pushs  argv        ; Array of argument strings
                    @pushs  rc          ; RC from function called
                    @pushs  ret         ; Storage for returned data
                    @pushs  qname       ; Name of active data queue
                    call far ptr RXFUNCTIONCALL
                    endm
;
endif ;INCL_RXFUNC
;
```

RxFunctionDeregister – Remove Function

This call allows the user to remove a function.

RxFunctionDeregister (FuncName)

Parameters

FuncName *(PSZ)* – input
A pointer to the ASCIIZ string containing the name of the external function to remove.

Returns

0 RXFUNC_OK
30 RXFUNC_NOTREG

C Language

```
/***    RxFunctionDeregister - Delete a function from the AFT */

USHORT APIENTRY RxFunctionDeregister (
       PSZ );                        /* Name of function to remove */
```

Assembler Language

```
;*  RxFunctionDeregister - Delete a function from the AFT
;
;*  @RxFunctionDeregister *Name
;
@RxFunctionDeregister macro name
                    @define RXFUNCTIONDEREGISTER
                    @pushs  name        ;  Name of function to remove
                    call far ptr RXFUNCTIONDEREGISTER
                    endm
;
```

RxFunctionQuery – Query List of Functions

This call allows the user to query the list of available functions within a particular function.

RxFunctionQuery (FuncName)

Parameters

FuncName *(PSZ)* – input
A pointer to the ASCIIZ string containing the name of the external function to search for.

Returns

0	RXFUNC_OK
30	RXFUNC_NOTREG

Remarks

This function will return **RXFUNC_OK** only if the requested function is available for calling by the current process. If a function is found with a registered name of *FuncName*, but is tagged as belonging to a different process, the search for the function will continue as if no registration had been found.

C Language

```
/***    RxFunctionQuery - Scan the AFT for a function */

USHORT APIENTRY RxFunctionQuery (
       PSZ );                        /* Name of function to find   */
```

Assembler Language

```
;*  RxFunctionQuery - Scan the AFT for a function
;
;*  @RxFunctionQuery *Name
;
@RxFunctionQuery    macro name
                    @define RXFUNCTIONQUERY
                    @pushs  name        ;  Name of function to find
                    call far ptr RXFUNCTIONQUERY
                    endm
;
```

RxFunctionRegister – Register Function

This call allows the user to register a function either as a dynalink procedure or as an address in a program in memory.

RxFunctionRegister (FuncName, ModuleName, EntryPoint, ModuleType)

Parameters

FuncName *(PSZ)* – input
A pointer to the ASCIIZ string containing the name of the external function.

ModuleName *(PSZ)* – input
A pointer to the ASCIIZ string containing the name of the dynamic link library for the function.

EntryPoint *(PSZ)* – input
A pointer to the ASCIIZ string containing the name of the dynalink procedure to perform the function.

ModuleType *(USHORT)* – input
A flag indicating in what type of module the function resides. The valid registration-type identifiers are:

RXFUNC_DYNALINK
indicates that the function is available in a dynamic link library.

RXFUNC_CALLENTRY
indicates that the function is registered as an entry point in memory.

Returns

0	RXFUNC_OK
10	RXFUNC_DEFINED
20	RXFUNC_NOMEM

Remarks

When *ModuleType* is **RXFUNC_CALLENTRY**, a function address in the current program is registered. Because the third parameter is defined as a PSZ, the address of the function will have to be cast on the call. The *ModuleName* parameter is ignored for this type of registration.

C Language

```
/***    RxFunctionRegister - Register a function in the AFT */

USHORT APIENTRY RxFunctionRegister (
       PSZ,                          /* Name of function to add    */
       PSZ,                          /* Dll file name (if in dll)  */
       PSZ,                          /* Entry in dll OR mem address*/
       USHORT);                      /* RX_DYNALINK || RX_CALLENTRY*/
```

Assembler Language

```
;*  RxFunctionRegister - Register a function in the AFT
;
;*  @RxFunctionRegister *name,*dllname,*funcname,type
;
@RxFunctionRegister macro name,dll,fn,type
                    @define RXFUNCTIONREGISTER
                    @pushs  name        ;  Name of function to add
                    @pushs  dll         ;  Dll file name (if in dll)
                    @pushs  fn          ;  Entry in dll OR mem address
                    @pushw  type        ;  RX_DYNALINK || RX_CALLENTRY
                    call far ptr RXFUNCTIONREGISTER
                    endm
;
```

Chapter 5. Macrospace Interface

The macrospace interface can reduce the search time for REXX functions on disk by allowing a function image to be maintained in memory for immediate load and execution. This capability is especially useful for procedures and functions which are used frequently in a program, such as editor macros.

For the REXX programmer, calling a macrospace-registered function is essentially identical to that for any other function call. The macrospace functions are global to all REXX procedures. The sole difference in execution is the priority is given in the external-function search order. Functions registered in the macrospace can be placed in this search order to be executed before or after all other external functions.

The REXX interpreter environment during the execution of macrospace functions is the same as if the functions existed in external REXX files.

Search Order

When a function is registered in the macrospace through the interface functions, a flag is passed requesting the function's position in the external function search order for the REXX interpreter. There are two valid requests for search order positioning: before all other external functions and after all other external functions.

Pre-External Function Registration

If a function is registered with the **RXMACRO_SEARCH_BEFORE** flag, the REXX interpreter locates that function **before** any functions existing in external REXX files or in function packages. This allows the user to supersede termporarily the normal execution of a function by replacing an external function without deleting or renaming the file or cancelling the function's registration in function packages.

Post-External Function Registration

If a function is registered with the **RXMACRO_SEARCH_AFTER** flag, the REXX interpreter locates that function will be located by the REXX interpreter **after** any functions existing in external REXX files or in function packages. This allows the user to provide a default function for the REXX interpreter to use if some external function required by the interpreter cannot be found in an external REXX file or function packages.

Storage of Macrospace Libraries

Functions registered in the macrospace can be saved to a permanent storage file on disk for loading at a later date. This allows an application, such as an editor, to create its own library of frequently used functions and load the entire macrospace library into memory for fast access by the language processor. This macrospace library is specified by a file name. Thus, multiple macrospace function libraries can exist concurrently on a disk.

Some consideration must be given to matters of storage and memory utilization. The fact that the macrospace interface is designed to enhance performance for external REXX functions places severe limitations on the number of restrictions imposed by the API. Therefore, the only limitations placed on the user are the constraints of available memory imposed by OS/2. The macrospace may grow as long as memory is available. However, if the macrospace grows too large (relative to the amount of available memory), performance of the operating system may deteriorate due to an increased number of disk accesses to maintain its swap file.

Therefore, the macrospace should not be used to hold all of the functions needed by the user for daily normal use. A more efficient method is to maintain only those functions which the current process uses in the macrospace, and to either delete or omit others from the macrospace.

Macrospace Interface Calls

The calls that you use for registering and manipulating functions with the macrospace interface are:

- RxMacroChange (Change or Add Function to Macrospace)
- RxMacroDrop (Delete Function from Macrospace)
- RxMacroErase (Erase Functions in Macrospace)
- RxMacroLoad (Load Functions into Macrospace)
- RxMacroQuery (Search for Function in Macrospace)
- RxMacroReOrder (Change Search Order in Macrospace)
- RxMacroSave (Save Functions in Macrospace).

The calls are described on the following pages.

RxMacroChange – Change or Add Function to Macrospace

This call allows the user to change or add a function to the macrospace.

RxMacroChange (FuncName, SourceFile, Position)

Parameters

FuncName *(PSZ)* – input
A pointer to the ASCIIZ string specifying the name by which the function will be known. This function name must be validated to follow the REXX rules for function names; it should also correspond to function names implemented as part of external function packages.

SourceFile *(PSZ)* – input
A pointer to the ASCIIZ string file specification of the disk file containing the source for this function. If no file extension is supplied it defaults to ".CMD." If the full path is not specified the standard search is conducted (search current directory and OS/2 path).

Position *(USHORT)* – input
The flag indicating where in the external function search order for REXX this function should be placed. There are two possible positions: a function can be placed at the beginning of the search order (that is, before all other external functions), or at the end of the search order (that is, after all other external functions).

Returns

0 RXMACRO_OK
1 RXMACRO_NO_STORAGE
7 RXMACRO_SOURCE_NOT_FOUND
8 RXMACRO_INVALID_POSITION

C Language

```
/***    RxMacroChange - Register a function in the Macro Space */

USHORT APIENTRY RxMacroChange (
        PSZ,                        /* Function to add/change      */
        PSZ,                        /* Name of file to get function*/
        USHORT );                   /* Flag indicating search pos  */
```

Assembler Language

```
;*  RxMacroChange - Register a function in the Macro Space
;
;*  @RxMacroChange *Name,*Filename,flag
;
@RxMacroChange      macro name,fname,flag
                    @define RXMACROCHANGE
                    @pushs  name      ; Function to add/change
                    @pushs  fname     ; Name of file to get
                    @pushw  flag      ; Flag indicating search pos
                    call far ptr RXMACROCHANGE
                    endm
;
```

RxMacroDrop – Delete Function from Macrospace

This call allows the user to delete a function from the macrospace.

RxMacroDrop (FuncName)

Parameters

FuncName *(PSZ)* – input
A pointer to the ASCIIZ string specifying the name of the function to remove.

Returns

0 RXMACRO_OK
2 RXMACRO_NOT_FOUND

C Language

```
/***    RxMacroDrop - Remove a function from the Macro Space */

USHORT APIENTRY RxMacroDrop (
        PSZ );                      /* Name of function to remove */
```

Assembler Language

```
;*  RxMacroDrop - Remove a function from the Macro Space
;
;*  @RxMacroDrop *Name
;
@RxMacroDrop        macro   name
                    @define RXMACRODROP
                    @pushs  name      ; Name of function to remove
                    call far ptr RXMACRODROP
                    endm
;
```

RxMacroErase – Erase Functions in Macrospace

This call allows the user to erase all functions in the macrospace.

RxMacroErase ()

Returns

0	RXMACRO_OK
2	RXMACRO_NOT_FOUND

Remarks

There are no parameters to this function. Special care should be taken in using this function, because no verification is done on the deleted functions; that is, all functions are removed from the macrospace even if other processes use that function.

C Language

```
/***    RxMacroErase - Remove all functions from the Macro Space */

USHORT APIENTRY RxMacroErase(
        VOID );                     /* No Arguments.              */
```

Assembler Language

```
;*  RxMacroErase - Remove all functions from the Macro Space
;
;*  @RxMacroErase
;
@RxMacroErase       macro
                    @define RXMACROERASE
                    call far ptr RXMACROERASE
                    endm
;
```

RxMacroLoad –
Load Functions into Macrospace

This call allows the user to load either all functions or a subset of all functions from a file into the macrospace.

RxMacroLoad (FuncCount, FuncNames, MacroLibFile)

Parameters

FuncCount *(USHORT)* – input
The number of functions requested for loading from the source file.

FuncNames *(PSZ FAR *)* – input
A pointer to a list of ASCIIZ strings containing the names of the functions to be loaded from the file (number of strings in list given by *FuncCount*).

MacroLibFile *(PSZ)* – input
A pointer to the ASCIIZ file name string to load workspace from.

Returns

0	RXMACRO_OK
1	RXMACRO_NO_STORAGE
2	RXMACRO_NOT_FOUND
4	RXMACRO_ALREADY_EXISTS
5	RXMACRO_FILE_ERROR
6	RXMACRO_SIGNATURE_ERROR

Remarks

If *FuncCount* is zero or *FuncNames* is NULL, all of the functions in the file will be loaded.

The *FuncNames* parameter is a pointer to a list of pointers, similar to the *argv* parameter list in the **main()** function of a C-language program.

If a request is made to load a function which already exists (that is, if there is already a function with the same name in the macrospace), the entire load request is discarded and the macrospace remains unchanged.

C Language

```
/***    RxMacroLoad - Load Macro Space functions from a file */

USHORT APIENTRY RxMacroLoad (
        USHORT,                     /* Argument count (0==load all)*/
        PSZ FAR *,                  /* List of funct names to load */
        PSZ );                      /* File to load functions from */
```

Assembler Language

```
;*  RxMacroLoad - Load Macro Space functions from a file
;
;*  @RxMacroLoad Argc,**Listofnames,*Filename
;
@RxMacroLoad        macro   argc,lname,fname
                    @define RXMACROLOAD
                    @pushw  argc      ; Argument count (0==load)
                    @pushs  lname     ; List of funct names to load
                    @pushs  fname     ; File to load functions from
                    call far ptr RXMACROLOAD
                    endm
;
```

RxMacroQuery — Search for Function in Macrospace

This call allows the user to search for a function in the macrospace.

RxMacroQuery (FuncName, Position)

Parameters

FuncName *(PSZ)* — input
A pointer to the ASCIIZ string specifying the name of the function to seek in the list of workspace functions.

Position *(PUSHORT)* — output
A pointer to a USHORT for return of the current search-order position of the function, if it exists. If the function does not exist in the macrospace, the value will not be changed.

Returns

0 RXMACRO_OK
2 RXMACRO_NOT_FOUND

C Language

```
/***    RxMacroQuery - Find a function's search-order position */

USHORT APIENTRY RxMacroQuery (
        PSZ,                        /* Function to search for      */
        PUSHORT );                  /* Ptr for position flag return*/
```

Assembler Language

```
;*  RxMacroQuery - Find a function's search-order position
;
;*  @RxMacroQuery *Name,*Flag
;
@RxMacroQuery       macro   name,flag
                    @define RXMACROQUERY
                    @pushs  name      ; Function to search for
                    @pushs  flag      ; Ptr for position flag
                    call far ptr RXMACROQUERY
                    endm
;
```

RxMacroReOrder — Change Search-Order in Macrospace

This call allows the user to change the search-order location of a function in the macrospace.

RxMacroReOrder (FuncName, Position)

Parameters

FuncName *(PSZ)* — input
A pointer to the ASCIIZ string specifying the name of the function whose search order positioning is to be changed.

Position *(USHORT)* — input
The flag indicating where this function is placed in the external function search order for REXX.

Returns

0 RXMACRO_OK
2 RXMACRO_NOT_FOUND
8 RXMACRO_INVALID_POSITION

C Language

```
/***    RxMacroReOrder - Change a function's search-order position */

USHORT APIENTRY RxMacroReOrder(
        PSZ,                        /* Name of funct change order  */
        USHORT );                   /* New position for function   */
```

Assembler Language

```
;*  RxMacroReOrder - Change a function's search-order position
;
;*  @RxMacroReOrder *Name,Position
;
@RxMacroReOrder     macro   name,pos
                    @define RXMACROREORDER
                    @pushs  name      ; Name of funct change order
                    @pushw  pos       ; New position for function
                    call far ptr RXMACROREORDER
                    endm
;
```

Search Order Flags

The following are the flags which are passed to the macrospace interface to specify where a function should be placed in the REXX search order for external functions:

RXMACRO_SEARCH_BEFORE place function at top of search order
RXMACRO_SEARCH_AFTER place function at bottom of search order.

RxMacroSave – Save Functions in Macrospace

This call allows the user to save either all functions or a subset of all functions in the macrospace to a file.

RxMacroSave (FuncCount, FuncNames, MacroLibFile)

Parameters

FuncCount *(USHORT)* – input
The number of functions requested for saving to the file.

FuncNames *(PSZ FAR *)* – input
A pointer to a list of ASCIIZ strings containing the names of the functions to be saved to the file (number of strings in list given by *FuncCount*).

MacroLibFile *(PSZ)* – input
A pointer to the ASCIIZ filename string to save the workspace to. If file exists it is overwritten. If file does not exist it is created.

Returns

0	RXMACRO_OK
2	RXMACRO_NOT_FOUND
3	RXMACRO_EXTENSION_REQUIRED
5	RXMACRO_FILE_ERROR

Remarks

If *FuncCount* is zero or *FuncNames* is NULL, all of the functions in the macro space are saved.

The *FuncNames* parameter is a pointer to a list of pointers, similar to the *argv* parameter list in the **main()** function of a C program.

C Language

```
/***    RxMacroSave - Save Macro Space functions to a file */

USHORT APIENTRY RxMacroSave (
        USHORT,                     /* Argument count (0==save all)*/
        PSZ FAR *,                  /* List of funct names to save */
        PSZ );                      /* File to save functions in   */
```

Assembler Language

```
;*  RxMacroSave - Save Macro Space functions to a file
;
;*  @RxMacroSave Argc,**Listofnames,*Filename
;
@RxMacroSave        macro   argc,lname,fname
                    @define RXMACROSAVE
                    @pushw  argc      ; Argument count (0==save)
                    @pushs  lname     ; List of funct names to save
                    @pushs  fname     ; File to save functions in
                    call far ptr RXMACROSAVE
                    endm
;
```

Chapter 6. Variable Pool Interface

Applications use the variable pool interface to manipulate the variable pool of the currently active REXX procedure. Here are C- and Assembler-language examples of variable pool APIs.

C Language

```
/***    Shared Variable Pool Interface */

/***    Function Codes for Variable Pool Interface (shvcode) */

#define RXSHV_SET          0x00        /* Set variable from given value     */
#define RXSHV_FETCH        0x01        /* Copy value of variable to buffer  */
#define RXSHV_DROPV        0x02        /* Drop variable                     */
#define RXSHV_SYSET        0x03        /* Symbolic name Set variable        */
#define RXSHV_SYFET        0x04        /* Symbolic name Fetch variable      */
#define RXSHV_SYDRO        0x05        /* Symbolic name Drop variable       */
#define RXSHV_NEXTV        0x06        /* Fetch "next" variable             */
#define RXSHV_PRIV         0x07        /* Fetch private information          */
#define RXSHV_EXIT         0x08        /* Set function exit value           */


/***    Return Code Flags for Variable Pool Interface (shvret) */

#define RXSHV_OK           0x00        /* Execution was OK                  */
#define RXSHV_NEWV         0x01        /* Variable did not exist            */
#define RXSHV_LVAR         0x02        /* Last var transferred via SHVNEXTV */
#define RXSHV_TRUNC        0x04        /* Truncation occurred during Fetch  */
#define RXSHV_BADN         0x08        /* Invalid variable name             */
#define RXSHV_MEMFL        0x10        /* Out of memory failure             */
#define RXSHV_BADF         0x80        /* Invalid function code (shvcode)   */


/***    Structure of Shared Variable Request Block (SHVBLOCK) */

typedef struct shvnode {
    struct shvnode FAR *shvnext;       /*  Chain pointer to the next block   */
    RXSTRING           shvname;        /*  Pointer to the variable name      */
    RXSTRING           shvvalue;       /*  Pointer to the value buffer       */
    ULONG              shvnamelen;     /*  Length of the name value          */
    ULONG              shvvaluelen;    /*  Length of the fetch value         */
    UCHAR              shvcode;        /*  Individual function code          */
    UCHAR              shvret;         /*  Individual return code flags      */
    } SHVBLOCK;


typedef SHVBLOCK FAR *PSHVBLOCK;


SHORT APIENTRY RxVar (
        PSHVBLOCK);    /* Pointer to a list of SHVBLOCK's             */

#endif /* INCL_RXSHV */
```

Assembler Language

```
;***    Shared Variable Pool Interface
ifdef INCL_RXSHV
;
;***    Function Codes for Variable Pool Interface (shvcode)
;
RXSHV_SET           equ 0000h          ; Set var from given value
RXSHV_FETCH         equ 0001h          ; Copy value of var to buffer
RXSHV_DROPV         equ 0002h          ; Drop variable
RXSHV_SYSET         equ 0003h          ; Symbolic name Set variable
RXSHV_SYFET         equ 0004h          ; Symbolic name Fetch variable
RXSHV_SYDRO         equ 0005h          ; Symbolic name Drop variable
RXSHV_NEXTV         equ 0006h          ; Fetch "next" variable
RXSHV_PRIV          equ 0007h          ; Fetch private information
RXSHV_EXIT          equ 0008h          ; Set function exit value
;
;
;***    Return Code Flags for Variable Pool Interface (shvret)
;
RXSHV_OK            equ 0000h          ; Execution was OK
RXSHV_NEWV          equ 0001h          ; Variable did not exist
RXSHV_LVAR          equ 0002h          ; Last var trans via SHVNEXTV
RXSHV_TRUNC         equ 0004h          ; Truncation occurred-Fetch
RXSHV_BADN          equ 0008h          ; Invalid variable name
RXSHV_MEMFL         equ 0010h          ; Out of memory failure
RXSHV_BADF          equ 0080h          ; Invalid funct code
RXSHV_NOAVL         equ 0090h          ; Interface not available
;
;
;***    Structure of Shared Variable Request Block (SHVBLOCK)
;
SHVBLOCK struc
shvnext             dd  ?              ; Pointer to the next block
shvname             dd  ?              ; Pointer to the name buffer
shvvalue            dd  ?              ; Pointer to the value buffer
shvnamelen          dd  ?              ; Length of the name value
shvvaluelen         dd  ?              ; Length of the fetch value
shvcode             db  ?              ; Function code for this block
shvret              db  ?              ; Individual Return Code Flags
SHVBLOCK ends
;
PSHVBLOCK           dd  ?
;
;
;*  RxVar - Request Variable Pool Service
;
;*  @RxVar *Shared Variable Request Block
;
@RxVar              macro   shv
                    @define RXVAR
                    @pushs  shv        ; Pointer to list of SHVBLOCKs
                    call far ptr RXVAR
                    endm
;
endif ;INCL_RXSHV
;
```

RxVar – Pass Request Blocks to Variable Pool

This call allows the user to pass a linked list of shared variable request blocks to the variable pool.

RxVar (RequestBlockList)

Parameters

RequestBlockList *(PSHVBLOCK)* – input

A pointer to a linked list of shared variable request blocks. Each request block can specify a different operation to be performed by the interpreter. Each request block is processed in turn; control returns to the caller after the last block or after a severe error. In this manner, the interpreter can perform multiple operations by issuing only one call.

An integer return code will be returned from the RxVar call and will contain the return code from the entire set of requests. Return codes for each individual shared variable request block is found in a field of the block.

Returns

0 or positive	Entire shared variable request block list was processed. The return code is the composite OR of the low-order 6 bits of the *shvret* bytes.
-1	Invalid entry conditions.
-2	Insufficient storage was available for a request set. Processing was aborted (some of the request blocks may remain unprocessed - their *shvret* bytes will be unchanged).
-3	The variable pool interface was not enabled when this call was issued.
127	The RxVar routine was not found. Equivalent to DosGetProcAddr error_proc_not_found.

C Language

```
/***    RxVar - Request Variable Pool Service */

USHORT APIENTRY RxVar(
        PSHVBLOCK);                /* Pointer to list of SHVBLOCKs*/

#endif /* INCL_RXSHV */
```

Assembler Language

```
;*  RxVar - Request Variable Pool Service
;
;*  @RxVar *Shared Variable Reqest Block
;
@RxVar              macro   shv
                    @define RXVAR
                    @pushs  shv        ; Pointer to list of SHVBLOCKs
                    call far ptr RXVAR
                    endm
;
```

Interface Types

Three of the variable pool interface functions — set, fetch, and drop — have dual interfaces, the symbolic and the direct.

Symbolic Interface

The symbolic interface refers to normal REXX substitution when interpreting variables. Variable names are valid REXX symbols (in mixed case if desired) including compound symbols. The functions that use the symbolic interface are **RXSHV_SYSET**, **RXSHV_SYFET**, and **RXSHV_SYDRO**.

Direct Interface

The direct interface uses no substitution or case translation. Simple symbols must be valid REXX variable names: in uppercase only and not starting with a digit or a period. Compound symbols, however, allow any characters permitted in symbols, including lowercase characters and blanks, following a valid REXX stem. The direct interface is used by **RXSHV_SET**, **RXSHV_FETCH**, **RXSHV_DROP**, **RXSHV_NEXTV**, **RXSHV_PRIV**, and **RXSHV_EXIT**.

Shared Variable Functions

This section describes the actions for each function code.

Notes:

1. The terms *value length*, *value pointer*, *name length* and *name pointer* will be used generically in the following definitions to describe information derived from the *shvvalue* and *shvname* **RXSTRING** structures.
2. In general terms, the **RXSTRING** structures *shvname* and *shvvalue* are used to describe the memory buffers being passed through the variable pool interface. The lengths *shvnamelen* and *shvvaluelen* are the actual lengths of data processed.

Setting Variables

RXSHV_SET and RXSHV_SYSET
The *name* pointer and length describes the name of the variable to be set, and the *value* pointer and length describes the value which is assigned to it. The name is validated to ensure that it does not contain invalid characters, and the variable is then set from the value given. If the name is a stem, all variables with that stem are set, just as though this was a REXX assignment. *RXSHV_NEWV* is set if the variable did not exist before the operation.

Fetching Variables

RXSHV_FETCH and RXSHV_SYFET
The *name* pointer and length describe a buffer containing the name of the variable to be fetched; *shvnamelen* is the length of the name within the buffer. The *value* pointer and length describe a buffer to receive the value of the specified variable. The name is validated to ensure that it does not contain invalid characters, and the variable is then located and copied to the buffer. The total length of the variable is put into *shvvaluelen*. If the value was truncated (because the buffer was not big enough) the *RXSHV_TRUNC* bit is set. If the variable is shorter than the length of the buffer, no padding takes place. If the name is a stem, the initial value of that stem (if any) is returned.

RXSHV_NEWV is set if the variable did not exist before the operation, and in this case the value copied to the buffer is the derived name of the variable, such as after substitution. See "Notes and Limits" on page 6-6 for additional information.

RXSHV_NEXTV
(Fetch next variable.) This function may be used to search through all the variables known to the interpreter (that is, all those of the current generation, excluding those "hidden" by **PROCEDURE** instructions). The order in which the variables are revealed is not specified.

The interpreter maintains a pointer to its list of variables. This is reset to point to the first variable in the list whenever a host command is issued, or any function other than **RXSHV_NEXTV** is executed by way of the variable pool interface.

Whenever a **RXSHV_NEXTV** function is executed the name and value of the next available variable are copied to two buffers supplied by the caller.

The *name* pointer and length describe a buffer into which the name is copied. The total length of the name is returned in *shvnamelen*, and if the name was truncated (because the buffer was not big enough) the *RXSHV_TRUNC* bit is set. If the name is shorter than the length of the buffer, no padding takes place. The value of the variable is copied to the user's buffer area using the same protocol as for the Fetch operation.

If *shvret* has *RXSHV_LVAR* set, the end of the list of known variables has been found, the internal pointers have been reset, and no valid data has been copied to the user buffers.

If *RXSHV_TRUNC* is set, either the name or the value has been truncated.

By repeatedly executing the *RXSHV_NEXTV* function (until the *RXSHV_LVAR* flag is set) a user program may locate all the REXX variables of the current generation. See "Notes and Limits" on page 6-6 for additional information.

RXSHV_PRIV
(Fetch private information.) This interface is similar to the **RXSHV_FETCH** interface, except that the name refers to certain fixed information items that are available. The entire name must be specified for the fetch to occur. The following names are recognized:

PARM The number of parameters (arguments) supplied to the procedure will be placed in the caller's buffer. The number will be formatted as a character string.

PARM.n The Nth parameter (argument) string will be placed in the caller's buffer. If the Nth parameter (argument) was not supplied to the program (whether omitted, null, or fewer than N parameters were specified), a null string will be returned.

Note: PARM.1 will return the same result as ARG.

QUENAME Fetch data queue name. The name of the currently active data queue is copied to the user's buffer.

SOURCE Fetch source string. The source string, as described for /REF37/, is copied to the user's buffer.

VERSION Fetch version string. The version string, as described for /REF38/, is copied to the user's buffer.

See "Notes and Limits" on page 6-6 for additional information.

Dropping Variables

RXSHV_DROPV and RXSHV_SYDRO
The *name* pointer and length describes the name of the variable to be dropped. The *value* pointer and length are not used. The name is validated to ensure that it does not contain invalid characters. The variable, if it exists, is dropped. If the name given is a stem, all variables starting with that stem are dropped.

Setting an Exit Return Value

RXSHV_EXIT
Set function or exit return value. An external function written in a language other than REXX or a system exit must have the capability of returning an exit value. A single call is allowed for each exit. The value passed on this call is treated like the expression of a RETURN instruction. This interface is identical to the **RXSHV_SET** interface, except that no name is specified. The **RXSHV_EXIT** interface is only valid during processing of an external function or a system exit that allows a string value to be returned.

Notes and Limits

- The full interface is enabled only during the execution of commands (including CMD subcommands), external routines (functions and subroutines), and some system exits (RXINIT, RXTER, RXCMD, and RXFNC). It is enabled for **RXSHV_EXIT** operations only from exits that allow a character string to be returned (RXHLT, RXMSQ, RXSIO). An attempt to call the RxVar entry point when it is not enabled will result in a return code of -3 (RxVar not enabled).
- On all fetch-type requests, it is allowable *not* to specify return buffers on the call and have the REXX interpreter allocate and give appropriate memory to the caller. On **RXSHV_FETCH**, **RXSHV_SYFET**, and **RXSHV_PRIV**, the value buffers may be left for REXX to allocate. On **RXSHV_NEXTV**, the value and the name buffers may be allocated by REXX.

  To have REXX supply name buffers, set *shvname* to an empty string for the call. REXX will allocate a memory segment, set up the name pointer and length, and set *shvnamelen* to the length of the name returned. This memory (that is, the far pointer stored in the *shvname* **RXSTRING**) must be freed by the user via DosFreeSeg.

  To have REXX supply value buffers set *shvvalue* to an empty string for the call. The REXX interpreter will allocate segments for the value buffer. The *value* pointer and length will describe segments as appropriate for the size of the buffer returned. The total length of the requested variable's contents is returned in *shvvaluelen*. This memory must be freed by the user via DosFreeSeg for the selector stored in the *shvvalue* **RXSTRING**.

  Note that by allowing REXX to supply these buffers a *RXSHV_TRUNC* condition will not occur. However, it is possible to get *RXSHV_MEMFL* errors for these operations. If *RXSHV_MEMFL* occurs no memory will have been allocated for that **SHV_BLOCK** operation.
- Only the main thread of a REXX application is allowed to access the REXX variable pool. Applications may create and use new threads in the course of performing their function, but only the original thread that started the application may interface to the REXX variable pool.

Chapter 7. System Exits

The system exits interface allows REXX to operate under a user-defined environment. When these exits are used, certain system-dependent interpreter activities can be executed as calls to routines provided by the caller of the interpreter, instead of by the interpreter itself. Except as provided for here, the interpreter does not allow the results of the program being interpreted to be affected by these exits.

There are exits for:

- The administration of resources, at the beginning and end of interpretation
- Assisted linkages to general classes of external facilities, such as functions
- Special language features, such as I/O to standard resources
- Polling for external interrupts and conditions.

As happens with subroutine and external-function handlers, system exit routines can be packaged either as dynalink libraries or as .EXE modules.

For details on the requirements and options for packaging API handlers, see "General Characteristics" on page 1-2.

Writing System Exit Handlers

This is an example of a system exit handler:

```
short pascal far my_sysexit (
     short func,      /* integer code defining the exit function     */
     short subfunc,   /* integer code defining the exit sub-function */
     char far *parm;  /* function dependent control block            */
```

Entry Conditions

All exits are passed the following three parameters on entry:

- The integer code for the exit being invoked
- The subfunction code for the exit
- A far pointer to the exit parameter list.

The parameter list pointed to by the third parameter contains information specific to the exit being invoked. See the exit descriptions that follow for the format of the parameter list for each exit. Note that some of the exits do not have any additional parameters. For those exits, the parameter list pointer will be null.

Exit Conditions

On return from the exit, the integer return value indicates the results of the exit call and the parameter list will have been updated appropriately. The value of the return code can signal one of the three following actions:

0 successful handling of the service. The parameter list has been updated as appropriate for that exit.

1 exit chooses not to handle the service request. The interpreter will handle the request through the default means.

-1 a fatal error occurred during processing of this request. REXX error 48, "Failure in system service," will be raised.

As an example, suppose you create a routine to process output from the SAY instruction:

- If your exit routine returns 1, SAY processing continues in the normal manner and the REXX interpreter displays the output string on the screen as usual.

- If your exit routine returns 0, the interpreter assumes that your exit routine has performed all required output. It therefore bypasses SAY processing and does not display the output string.
- If your exit routine returns -1, the interpreter halts with error 48, "Failure in system service."

For many of the exits, an arbitrary length character string may be passed back to the interpreter using the *SHVEXIT* function of the **RxVar** routine. This routine may only be called once for each invocation of the exit to return a value.

Identifying Handlers to REXX

System exit handlers must be registered with REXX by the RxExitRegister call before they may be used. This process is only used to give REXX an easy way to handle the routing of exit calls. The registration of system exit handlers is similar to the registration of subcommand handlers (the two processes both use the REXX SCBLOCK), but only the registration is similar. The calling conventions and handling of system exits is different from subcommands. See the REXXSAA.H file for C-language definitions and the REXXSAA.INC file for Assembler-language definitions of the RxExitRegister, RxExitQuery, and RxExitDrop functions.

Enabling System Exits

The interpreter exits are defined on invocation of the interpreter by the *sysexit* parameter of the call to the REXX interpreter. The *sysexit* parameter is a pointer to an array of **RXSYSEXIT** structures. Each **RXSYSEXIT** in the array consists of an INT code identifying the exit to be enabled and a pointer to an ASCIIZ string defining the name of a registered system exit environment that supplies the system exit routine handler. Create an extra **RXSYSEXIT** element and assign the code to **RXENDLST** to indicate the end of the system exit list.

System Exit Interface Calls

The calls for the system exit interface are similar to some calls for the subcommand environment interface. Chapter 3, Subcommand Interface describes the subcommand environment interface calls. The parameters and return codes are identical; only the call names are different. The following table compares the two sets of calls:

System Exit Call	Purpose	Subcommand Call	Read About It On
RxExitDrop	Allows user to drop a system exit environment	RxSubcomDrop	Page 3-4
RxExitQuery	Allows user to query a system exit environment name	RxSubcomQuery	Page 3-8
RxExitRegister	Allows user to register a system exit environment	RxSubcomRegister	Page 3-10

Exits

The following examples, in C and Assembler languages, show how exits may be specified in the list with the exit code and subcode definitions.

C Language

```
/* System Exit function and sub-function definitions */

#define RXENDLST    0       /* End of exit list.                      */
#define RXFNC    2          /* Process external functions.            */
#define    RXFNCCAL 1       /*    Currently the only subcode value.   */
#define RXCMD    3          /* Process host commands.                 */
#define    RXCMDHST 1       /*    Currently the only subcode value.   */
#define RXMSQ    4          /* Manipulate queue.                      */
#define    RXMSQPLL 1       /*    Pull a line from queue              */
#define    RXMSQPSH 2       /*    Place a line on queue               */
#define    RXMSQSIZ 3       /*    Return number of lines on queue     */
#define    RXMSQNAM 20      /*    Set active queue name (OS/2 only)   */

#define RXSIO    5          /* Session I/O.                           */
#define    RXSIOSAY 1       /*    SAY a line to STDOUT                */
#define    RXSIOTRC 2       /*    Trace output                        */
#define    RXSIOTRD 3       /*    Read from session char stream       */
#define    RXSIODTR 4       /*    DEBUG read from session char stream*/
#define    RXSIOTLL 5       /*    Return line length (N/A for OS/2)   */

#define RXHLT    7          /* Halt processing.                       */
#define    RXHLTCLR 1       /*    Clear HALT indicator                */
#define    RXHLTTST 2       /*    Test HALT indicator                 */
#define RXTRC    8          /* Test external trace indicator.         */
#define    RXTRCTST 1       /*    Currently the only subcode value.   */
#define RXINI    9          /* Initialization processing.             */
#define    RXINIEXT 1       /*    Currently the only subcode value.   */
#define RXTER   10          /* Termination processing.                */
#define    RXTEREXT 1       /*    Currently the only subcode value.   */

#define RXNOOFEXITS 11      /* 1 more than largest exit number.       */

typedef unsigned char far * PEXIT ; /* pointer to exit parameter block*/
```

Assembler Language

```
;*** System Exit function and sub-function definitions
;
RXENDLST            equ 0             ; End of exit list.
RXFNC               equ 2             ; Process external functions.
RXFNCCAL            equ 1             ; subcode value.
RXCMD               equ 3             ; Process host commands.
RXCMDHST            equ 1             ; subcode value.
RXMSQ               equ 4             ; Manipulate queue.
RXMSQPLL            equ 1             ; Pull a line from queue
RXMSQPSH            equ 2             ; Place a line on queue
RXMSQSIZ            equ 3             ; Return num of lines on
RXMSQNAM            equ 20            ; Set active queue name
RXSIO               equ 5             ; Session I/O.
RXSIOSAY            equ 1             ; SAY a line to STDOUT
RXSIOTRC            equ 2             ; Trace output
RXSIOTRD            equ 3             ; Read from char stream
RXSIODTR            equ 4             ; DEBUG read from char stream
RXSIOTLL            equ 5             ; Return linelength(N/A OS/2)
RXHLT               equ 7             ; Halt processing.
RXHLTCLR            equ 1             ; Clear HALT indicator
RXHLTTST            equ 2             ; Test HALT indicator
RXTRC               equ 8             ; Test ext trace indicator.
RXTRCTST            equ 1             ; subcode value.
RXINI               equ 9             ; Initialization processing.
RXINIEXT            equ 1             ; subcode value.
RXTER               equ 10            ; Termination processing.
RXTEREXT            equ 1             ; subcode value.
RXNOOFEXITS         equ 11            ; 1 + largest exit number.
;
PEXIT               dd  ?             ; ptr to exit parameter block
;
```

Exit Characteristics

Each exit has the following specific characteristics:

- The occasions when the exit is called, if provided.
- The default action to be taken when the exit is not provided.
- The parameter list for the exit. This parameter list is additional to the subcode, which is always passed.
- The action to be taken by the exit.
- The continuation actions to be taken by the interpreter after return from the exit.
- Whether the variable pool interface is enabled during the exit. (It is enabled only for RXCMD, RXFUNC, RXINI, and RXTER; disabled during processing of all other exits.)

For many of these exits, an arbitrary length character string may be passed back through an **RXSTRING** in the parameter block. REXX will initialize the **RXSTRING** to an empty string before calling the exit.

If an exit returns a value in its parameter block and through the *RXSHV_EXIT* function of the **RxVar** routine, REXX uses the value set by the call to **RxVar**. See "Setting an Exit Return Value" on page 6-6 for details about using RXVAR and the functions available for exits.

Exit Definitions

The remainder of this chapter defines the following exit services and their subfunctions:

- RXFNC (Process External Functions) exit service
  - RXFNCCAL (Process External Functions)
- RXCMD (Process Host Commands)
  - RXCMDHST (Process Host Commands)
- RXMSQ (Manipulate Queue) exit service
  - RXMSPQLL (Pull Line from Queue)
  - RXMSQPSH (Place Line on Queue)
  - RXMSQSIZ (Return Number of Lines in Queue)
  - RXMSQNAM (Set Name of Active Queue)
- RXSIO (Session I/O) exit service
  - RXSIOSAY (Output Result of SAY Instruction)
  - RXSIOTRC (TRACE Output Processing. Call to Output TRACE Results)
  - RXSIOTRD (Read From STDIN Stream)
  - RXSIODTR (Debug Read)
- RXHLT (Halt Processing) exit service
  - RXHLTCLR (Clear Halt Indicator)
  - RXHLTTST (Test Halt Indicator)
- RXTRC (Test External Trace Indicator) exit service
  - RXTRCTST (Test External Trace Indicator)
- RXINI (Initialization Processing) exit service
  - RXINIEXT (Initialization Processing)
- RXTER (Termination Processing) exit service
  - RXTEREXT (Termination Processing)

This service supports one subfunction, RXFNCCAL. The request code for this subfunction is specified by the second parameter.

RXFNCCAL — Process external functions

When called
When interpretation is about to call an external routine.

Default action
Call the routine.

Action
Call the routine, if possible.

Continuation
If indicated by the return conditions, raise error 40, 43 or 44. Resume interpretation.

C Language

```
/***    Subfunction RXFNCCAL - External Function Calls */

typedef struct {
   struct {
      unsigned rxfferr  : 1;          /* Invalid call to routine.    */
      unsigned rxffnfnd : 1;          /* Function not found.         */
      unsigned rxffsub  : 1;          /* Called as a subroutine if   */
                                      /* set.  Return values are     */
                                      /* optional for subroutines,   */
                                      /* required for functions.     */
   } rxfnc_flags ;

   PUCHAR           rxfnc_name;       /* Pointer to function name.   */
   USHORT           rxfnc_namel;      /* Length of function name.    */
   PUCHAR           rxfnc_que;        /* Current queue name.         */
   USHORT           rxfnc_quel;       /* Length of queue name.       */
   USHORT           rxfnc_argc;       /* Number of args in list.     */
   PRXSTRING        rxfnc_argv;       /* Pointer to argument list.   */
                                      /* List mimics argv list in    */
                                      /* REXXSAA -- array of         */
                                      /* RXSTRINGs.                  */
   RXSTRING         rxfnc_retc;       /* Return value.               */
} RXFNCCAL_PARM;
```

RXFNC –
Process External Functions

Assembler Language

```
;***    Subfunction RXFNCCAL - External Function Calls
;
; rxfferr             db  ?             * Invalid call to routine.
; rxffnfnd            db  ?             * Function not found.
; rxffsub             db  ?             * Called as a subroutine if
;                                       * set.  Return values are
;                                       * optional for subroutines,
;                                       * required for functions.
;
RXFNC_FIELDS        RECORD  rxfferr:1,rxffnfnd:1,rxffsub:1,pad:13
;
;
RXFNCCAL_PARM struc
rxfnc_flags         RXFNC_FIELDS 1     ;* same size as an unsigned.
rxfnc_name          dd  ?              ;* Pointer to function name.
rxfnc_namel         dw  ?              ;* Length of function name.
rxfnc_que           dd  ?              ;* Current queue name.
rxfnc_quel          dw  ?              ;* Length of queue name.
rxfnc_argc          dw  ?              ;* Number of args in list.
rxfnc_argv          dd  ?              ;* Pointer to argument list.
                                       ;* List mimics argv list in
                                       ;* REXXSAA -- array of
                                       ;* RXSTRINGs.
rxfnc_retc;         dd  ?              ;* Return value.
RXFNCCAL_PARM ends
;
```

Remarks

Upon entry to RXFNCCAL, the name of the function to be invoked is defined by *rxfnc_name* and *rxfnc_namel*. The arguments to the function are indicated by *rxfnc_argc* and *rxfnc_argv*. If the named routine is invoked by a CALL instruction (rather than as a function call), the flag *rxffsub* is set "on."

On return from the exit, values in *rxfnc_flags* indicate the status of the processing of the function. If neither *rxfferr* or *rxffnfnd* is "on," the routine has been successfully invoked and has run successfully. The error flags are checked only if the exit returns 0.

The flag *rxffnfnd* indicates that the exit could not locate the routine with the given name. The interpreter will give error 43, "Routine not found." The flag *rxfferr* indicates that the parameters supplied to the routine were invalid. The interpreter will give error 40, "Invalid call to routine."

Note: The variable pool interface is fully enabled during calls to the RXFNC exits. Use the *RXSHV_EXIT* function to return the value of the function named in *rxfnc_name*, or return the value through the *rxfnc_retc* field of the exit parameter list. This routine may only be called once for each invocation of the exit to return a value. If the routine was invoked as a function and a result was not returned, the interpreter will give error 44, "Function did not return data." If the routine was invoked as a subroutine, the returned result is optional.

This service supports one subfunction, RXCMDHST. The request code for this subfunction is specified by the second parameter.

RXCMDHST — Process host commands

When called
When interpretation is about to pass a command string to some external environment.

Default action
Pass the command to the external environment

Action
Pass the command to the external environment, if possible.

Continuation
If indicated by the results, raise the ERROR or FAILURE condition. Resume interpretation.

C Language

```
/***    Subfunction RXCMDHST -- Process Host Commands     */

typedef struct {
   struct {                              /* Condition flags            */
      unsigned rxfcfail : 1;             /* Command failed.  Trap with */
                                         /* CALL or SIGNAL on FAILURE. */
      unsigned rxfcerr  : 1;             /* Command ERROR occurred.    */
                                         /* Trap with CALL or SIGNAL on */
                                         /* ERROR.                     */
   } rxcmd_flags;
   PUCHAR           rxcmd_address;       /* Pointer to address name.   */
   USHORT           rxcmd_addressl;      /* Length of address name.    */
   PUCHAR           rxcmd_dll;           /* dll name for command.      */
   USHORT           rxcmd_dll_len;       /* Length of dll name.  0 ==> */
                                         /* .EXE file.                 */
   RXSTRING         rxcmd_command;       /* The command string.        */
   RXSTRING         rxcmd_retc;          /* Pointer to return code     */
                                         /* buffer.  User allocated.   */
} RXCMDHST_PARM;
```

RXCMD — Process Host Commands

Assembler Language

```
;***    Subfunction RXCMDHST -- Process Host Commands
;
;
; rxfcfail            db  ?           * Command failed.  Trap with
;                                     * CALL or SIGNAL on FAILURE.
; rxfcerr             db  ?           * Command ERROR occurred.
;                                     * Trap with CALL or SIGNAL on
;                                     * ERROR.
;
RXCMD_FIELDS        RECORD rxfcfail:1,rxfcerr:1,pad:14
;
;
RXCMDHST_PARM struc
rxcmd_flags         RXCMD_FIELDS 1      ;* Condition flags
rxcmd_address       dd  ?               ;* Pointer to address name.
rxcmd_addressl      dw  ?               ;* Length of address name.
rxcmd_dll           dd  ?               ;* dll name for command.
rxcmd_dll_len       dw  ?               ;* Length of dll name. 0==.EXE
rxcmd_command       dd  ?               ;* The command string.
rxcmd_retc          dd  ?               ;* Pointer to return code
                                        ;* buffer.  User allocated.
RXCMDHST_PARM ends
;
```

Remarks

On entry to the exit, *rxcmd_retc* defines a buffer that may be used to return a value to be used for the return code in character format (that is, numeric return codes must be formatted as a character string). The return code may be a non-numeric value. The parameter *rxcmd_command* contains the command string itself. The parameters *rxcmd_dll* and *rxcmd_dll_len* define the name of the dynalink library specified in the ADDRESS instruction.

The flags *rxfcfail* and *rxfcerr* are used by this exit to indicate that an ERROR or FAILURE condition has occurred. The exit controls the definition of what constitutes an ERROR or FAILURE of a command.

Note: The variable pool interface and all subcommand-registration functions are fully enabled during calls to the RXCMD exits.

This service supports four subfunctions: RXMSQPLL, RXMSQPSH, RXMSQSIZ, and RXMSQNAM. The second parameter specifies the subfunction request code. The parameter list depends on the particular subfunction invoked.

RXMSQPLL — Pull a line from the queue

When called
Used to interpret PULL.

Default action
Pull from the current default queue.

Action
Provide a result.

Continuation
Resume interpretation, using the result.

C Language

```
/***     Subfunction RXMSQPLL -- Pull Entry from Queue */

typedef struct {
   RXSTRING          rxmsq_retc;       /* Pointer to dequeued entry   */
                                       /* buffer.  User allocated.    */
} RXMSQPLL_PARM;
```

Assembler Language

```
;***     Subfunction RXMSQPLL -- Pull Entry from Queue
;
RXMSQPLL_PARM struc
rxmsq_retc          dd  ?             ;* Pointer to dequeued entry
                                      ;* buffer.  User allocated.
RXMSQPLL_PARM ends
;
```

Remarks

This exit is invoked by the PULL and PARSE PULL instructions. Upon entry to RXMSQPLL, *rxmsq_retc* defines a buffer that may be used to return a value to be used for the line removed from the queue.

RXMSQPSH — Place a line on the queue

When called
Used to interpret PUSH and QUEUE.

Default action
Push the value on the current default queue.

Action
Push the value on some queue where RXMSQPLL subsequently can fetch it.

Continuation
Resume interpretation.

RXMSQ — Manipulate Queue

C Language

```
/***    Subfunction RXMSQPSH -- Push Entry on Queue */

typedef struct {
   struct {                             /* Operation flag             */
      unsigned rxfmlifo : 1;            /* Stack entry LIFO if set,   */
                                        /* FIFO if reset.             */
   } rxmsq_flags;

   RXSTRING          rxmsq_value;      /* The entry to be pushed.    */
} RXMSQPSH_PARM;
```

Assembler Language

```
;***    Subfunction RXMSQPSH -- Push Entry on Queue
;
;
; rxfmlifo            db  ?           * Stack entry LIFO if set,
;                                     * FIFO if reset.
;
RXMSQ_FIELDS        RECORD rxfmlifo:1,pad:15
;
;
RXMSQPSH_PARM struc
rxmsq_flags         RXMSG_FIELDS 1    ; Operation flag
rxmsq_value;        dd  ?             ; The entry to be pushed.
RXMSQPSH_PARM ends
;
```

Remarks

The line to be placed on the queue (*rxmsq_value*) is the result of evaluating the expression specified on a PUSH or QUEUE instruction. It is the responsibility of the exit to handle truncation of this string if the exit has a restriction on the maximum width of the queue. The stacking order is indicated by the flag *rxfmlifo*.

RXMSQSIZ — Return the number of lines in the queue

When called
Used to interpret the built-in function QUEUED.

Default action
Find the size of the current default queue.

Action
Return the size of the queue which RXMSQPLL pulls from.

Continuation
Use the returned value as the value of QUEUED. Resume interpretation.

C Language

```
/***    Subfunction RXMSQSIZ -- Return the Current Queue Size */

typedef struct {
   ULONG             rxmsq_size;     /* Number of Lines in Queue   */
} RXMSQSIZ_PARM;
```

Assembler Language

```
;***    Subfunction RXMSQSIZ -- Return the Current Queue Size
;
;
RXMSQSIZ_PARM struc
rxmsq_size          dd  ?             ;* Number of Lines in Queue
RXMSQSIZ_PARM ends
;
```

Remarks

This exit is invoked by the QUEUED function and by the PULL and PARSE PULL instructions.

On return, *rxmsq_size* contains the number of lines in the data queue as an integer.

RXMSQNAM — Set the name of the active queue

Used to interpret Rxqueue("SET", *newqueuename*) operating system specific routine.

Default action

Change the current default queue to *newname*.

Action

Change the queue to be operated upon by RXMSQPLL, RXMSQPSH, and RXMSQSIZ.

Continuation

Resume interpretation.

C Language

```
/***    Subfunction RXMXQNAM -- Set Current Queue Name */

typedef struct {
   PSZ              rxmsq_name;       /* Selector containing ASCIIZ  */
                                      /* queue name.  Change length  */
                                      /* with DosReallocSeg if       */
                                      /* required.                   */
} RXMSQNAM_PARM;
```

Assembler Language

```
;***    Subfunction RXMXQNAM -- Set Current Queue Name
;
;
RXMSQNAM_PARM struc
rxmsq_name         dw  ?              ;* Selector containing ASCIIZ
                                      ;* queue name.  Change length
                                      ;* with DosReallocSeg if
                                      ;* required.
RXMSQNAM_PARM ends
;
```

Remarks

This exit is invoked by the function **RXQUEUE("SET",***newqueuename***)**.

RXSIO — Session I/O

This service supports four subfunctions: RXSIOSAY, RXSIOTRC, RXSIOTRD, and RXSIODTR. The second parameter specifies the subfunction request code. The parameter list depends on the particular subfunction invoked.

RXSIOSAY — Output result of SAY to standard output stream

When called
To output the result of a SAY instruction.

Default action
Write to the session standard output device.

Action
Send the line.

Continuation
Resume interpretation.

C Language

```
/***    Subfunction RXSIOSAY -- Perform SAY Clause */

typedef struct {
   RXSTRING          rxsio_string;     /* String to display.         */
   } RXSIOSAY_PARM;
```

Assembler Language

```
;***    Subfunction RXSIOSAY -- Perform SAY Clause
;
;
RXSIOSAY_PARM struc
rxsio_string        dd  ?             ;* String to display.
RXSIOSAY_PARM ends
;
```

Remarks

The line to be delivered to the standard output device is the result of evaluating the expression specified on a SAY instruction. This string may be any length up to the size of the standard output device (as determined by default system processing). It is the responsibility of the exit to handle truncation of this string if the string longer that the standard output device.

RXSIOTRC — TRACE output processing/call to output TRACE results

When called
To output the result of each line of tracing.

Default action
Write to the session standard error device.

Action
Send the line.

Continuation
Resume interpretation.

C Language

```
/***    Subfunction RXSIOTRC -- Write Trace Output */

typedef struct {
  RXSTRING          rxsio_string;     /* Trace line to display.      */
} RXSIOTRC_PARM;
```

Assembler Language

```
;***    Subfunction RXSIOTRC -- Write Trace Output
;
;
RXSIOTRC_PARM struc
rxsio_string        dd  ?             ;* Trace line to display.
RXSIOTRC_PARM ends
;
```

Remarks

The line to be displayed at the standard error-output device is the result of a traced line. The form of the string contained in *rxsio_string* is:

Characters 1 though 6: the line number (with leading spaces).
A space, followed by the traced program line, ended by a semicolon.

RXSIOTRD — Read from STDIN stream

When called
To read the standard input stream (STDIN). Note that if the 'PULL' was satisfied by something from the queue, this routine is not called.

Default action
Read from the default character input stream.

Action
Read a line.

Continuation
Resume interpretation

C Language

```
/***    Subfunction RXSIOTRD -- Read Input from the Terminal */

typedef struct {
   RXSTRING          rxsiotrd_retc;    /* RXSTRING for output.  Note: */
                                       /* user allocates output       */
                                       /* buffer with DosAllocSeg()   */
                                       /* or DosAllocHuge().          */
} RXSIOTRD_PARM;
```

Assembler Language

```
;***    Subfunction RXSIOTRD -- Read Input from the Terminal
;
;
RXSIOTRD_PARM struc
rxsiotrd_retc       dd  ?             ;* RXSTRING for output.  Note:
                                      ;* user allocates output
                                      ;* buffer with DosAllocSeg()
                                      ;* or DosAllocHuge().
RXSIOTRD_PARM ends
;
```

Remarks

This exit is invoked by the instructions PULL and PARSE PULL when the active data queue is empty. (The PARSE LINEIN instruction does not invoke this exit.)

Upon entry, *rxsio_retc* defines a **RXSTRING** that may be used to return a value to be used for the line read from STDIN. REXX initializes this field to be a **RXSTRING** having a 250-character buffer. If additional length is needed, the system exit must allocate the output buffer with DosAllocSeg or DosAllocHuge.

RXSIODTR — Debug read

When called
To read from the session character stream when for interactive debug. This routine differs from RXSIOTRD because it is not dependent on the state of the queue.

Default action
Read from the default character input stream.

Action
Read a line.

Continuation
Resume interpretation

C Language

Debug read from STDIN: stream.

```
/***    Subfunction RXSIODTR -- Read Debug Input from the Terminal */

typedef struct {
   RXSTRING          rxsiotrd_retc;     /* RXSTRING for output. Note: */
                                        /* user allocates output      */
                                        /* buffer with DosAllocSeg()  */
                                        /* or DosAllocHuge().         */
} RXSIODTR_PARM;
```

Assembler Language

```
;***    Subfunction RXSIODTR -- Read Debug Input from the Terminal
;
RXSIODTR_PARM struc
rxsiodtr_retc       dd  ?           ;* RXSTRING for output.  Note:
                                       ;* user allocates output
                                       ;* buffer with DosAllocSeg()
                                       ;* or DosAllocHuge().
RXSIODTR_PARM ends
;
```

Remarks

This exit is invoked by STDIN input during interactive debugging (invoked by the TRACE instruction).

Upon entry, *rxsio_retc* defines a **RXSTRING** that may be used to return a value to be used for the line read from the STDIN stream. REXX initializes this field to a **RXSTRING** having a 250-character buffer. If additional length is needed, the system exit must allocate the output buffer with DosAllocSeg or DosAllocHuge.

This service supports two subfunctions: RXHLTCLR and RXHLTTST. The second parameter specifies the request code for each subfunction. The parameter list depends on the particular subfunction invoked.

RXHLTCLR — Clear halt indicator

This exit subfunction has no inputs or outputs. It is invoked by the interpreter to reset the external flag after RXHLTTST has returned a flag value of 1.

When called
After raising HALT and before the next (potential) call of RXHLTTST.

Default action
The interpreter makes use of the system facilities for resetting the polling of external interrupts.

Action
Reset the polling of the external interrupt.

Continuation
Continue interpretation

RXHLTTST — Test halt indicator

When called
RXHLTTST is called by the interpreter to poll whether an external attempt has been made to interrupt the execution. The interpreter must make enough calls to ensure that any execution that could be halted by this mechanism is eventually halted.

Default action
The interpreter makes use of the system facilities for polling external interrupts.

Action
Set the result to indicate whether the external interrupt has occurred.

Continuation
Raise the HALT condition if the interrupt has occurred. Continue interpretation.

C Language

```
/***    Subfunction RXHSTTST -- Test for HALT Condition */

typedef struct {
   struct {                          /* Halt flag                  */
      unsigned rxfhhalt : 1;              /* Set if HALT occurred.       */
   } rxhlt_flags;
} RXHLTTST_PARM;
```

Assembler Language

```
;***    Subfunction RXHSTTST -- Test for HALT Condition
;
;
; rxfhhalt            db  ?           * Set if HALT occurred.
;
RXHLT_FIELDS        RECORD rxfhhalt:1,pad:15
;
;
RXHLTTST_PARM struc
rxhlt_flags         RXHLT_FIELDS 1     ;* Halt flag
RXHLTTST_PARM ends
;
```

RXHLT – Halt Processing

Remarks

This exit informs the REXX interpreter when a HALT condition has been raised. It is invoked at the end of each clause. Upon return, the *rxfhhalt* flag will indicate whether a HALT condition has occurred (1 = Halt, 0 = No Halt). RXHLTTST may also return a string that is available through the built-in function CONDITION(D). This string is returned by using the *SHVEXIT* function of the RXVAR routine. This routine may only be called once for each invocation of the exit to return a value.

This service supports one subfunction: RXTRCTST. The request code for the subfunction is specified by the second parameter.

RXTRCTST — Test external trace indicator

When called

The tracing feature of REXX can be controlled externally to the program. RXTRC is called by the interpreter to poll whether the external indication says that trace is active.

Default action

The interpreter makes use of the system facilities for indicating this condition.

Action

Set the result to indicate whether the external condition is set.

Continuation

Continue with or without tracing, according to the result.

C Language

```
/***    Subfunction RXTRCTST -- Test for TRACE Condition */

struct rxtrc_parm {
  struct {
    unsigned rxftrace : 1;        /* External trace setting        */
         } rxtrc_flags;
}
```

Assembler Language

```
;***    Subfunction RXTRCTST -- Test for TRACE Condition
;
;
; rxftrace            db  ?            * Set to run external trace.
;
RXTRC_FIELDS        RECORD rxftrace:1,pad:15
;
;
RXTRCTST_PARM struc
rxtrc_flags         RXTRC_FIELDS 1     ;* Trace flags
RXTRCTST_PARM ends
;
```

Remarks

This exit is invoked at the end of each clause. Upon return from this exit, *rxftrace* will indicate whether an external trace condition has occurred (1 = External Trace, 0 = No External Trace).

Note: If the *rxftrace* flag is set from 0 to 1, on return, REXX will begin interactive debug mode. When the *rxftrace* flag is set from 1 to 0, trace will turn off interactive debug mode.

RXINI — Initialization Processing

This service supports one subfunction, RXINIEXT. The second parameter specifies the request code for the subfunction.

RXINIEXT — Initialization processing

This exit has no inputs or outputs. It is called before the first instruction of the program is interpreted.

Note: The variable pool interface is enabled when this exit is called.

When called
After the last instruction of the subject program is interpreted.

Default action
Nothing.

Action
This is an opportunity for the system to adapt to the change of context; for example, to initialize so that the other user exits will work.

Continuation
Resume interpretation

This service supports one subfunction, RXTEREXT. The second parameter specifies the request code for this subfunction.

RXTEREXT — Termination processing

This exit has no inputs or outputs. It is called after the last instruction of the program is interpreted. If the termination exit fails (that is, returns -1), no action is taken.

Note: The variable pool interface is enabled when this exit is called.

When called
After the last instruction of the subject program is interpreted.

Default action
Do nothing.

Action
This is an opportunity for the system to adapt to the change of context.

Continuation
Resume, leaving the interpreter.

Asynchronous Request Interface

Using the system exits for signaling halt and trace conditions causes a noticeable loss in performance. This loss is caused by the interpreter calling the exit routines for every REXX instruction. However, on most calls no action is taken by the exit routines.

The asynchronous request interface provides a similar capability to the system exits for halt and trace conditions without causing a loss of performance. The following three APIs allow an application to directly signal REXX for a trace or halt event.

RxHaltSet – Inform Halt Condition

This call allows the user to inform the REXX interpreter that a halt condition has occurred.

RxHaltSet (ProcessId, ThreadId)

Parameters

ProcessId (*PID*) – input
A pointer to the process ID that the REXX interpreter is running under. This is the same process as the program that issued the call to the REXXSAA DLL routine.

ThreadId (*TID*) – input
The thread ID that the REXX interpreter is running under. This is the same thread as the program that issued the call to the REXXSAA DLL routine.

Returns

0	RXARI_OK
1	RXARI_NOT_FOUND
2	RXARI_PROCESSING_ERROR

Remarks

This call will not be processed if the target REXX program is executing with the RXHLT exit enabled.

RxTraceSet – Turn On Debug Mode

This call allows the user to inform the REXX interpreter that the interpreter should turn on the interactive debug mode.

RxTraceSet (ProcessId, ThreadId)

Parameters

ProcessId (*PID*) – input
A pointer to the process ID that the REXX interpreter is running under. This is the same process as the program that issued the call to the REXXSAA DLL routine.

ThreadId (*TID*) – input
The thread ID that the REXX interpreter is running under. This is the same thread as the program that issued the call to the REXXSAA DLL routine.

Returns

0	RXARI_OK
1	RXARI_NOT_FOUND
2	RXARI_PROCESSING_ERROR

Remarks

This call will not be processed if the target REXX program is executing with the RXTRC exit enabled.

RxTraceReset — Turn Off Interactive Debug Mode

This call allows the user to inform the REXX interpreter that the interpreter should turn off the interactive debug mode.

RxTraceReset (ProcessId, ThreadId)

Parameters

ProcessId (*PID*) — input
A pointer to the process ID that the REXX interpreter is running under. This is the same process as the program that issued the call to the REXXSAA DLL routine.

ThreadId (*TID*) — input
The thread ID that the REXX interpreter is running under. This is the same thread as the program that issued the call to the REXXSAA DLL routine.

Returns

0	RXARI_OK
1	RXARI_NOT_FOUND
2	RXARI_PROCESSING_ERROR

Remarks

This call will not be processed if the target REXX program is executing with the RXTRC exit enabled.

Tracing is only turned off if the trace was originally turned on by a call to RxTraceSet.

Appendix. Errors Returned from REXX Calls

Numeric Order

Number	Name	Explanation
0x01	RXSUBCOM_ISREG	The subcommand environment has been registered.
0x01	RXSUBCOM_ERROR	An error in execution has occurred; the interpreter raises an ERROR condition.
0x02	RXSUBCOM_FAILURE	A failure in execution has occurred; the interpreter raises a FAILURE condition.
1002	RXSUBCOM_NOEMEM	There is insufficient memory available to complete this request.
1003	RXSUBCOM_BADTYPE	Incorrect registration type; an internal failure may have occurred. Retry the operation; if this error persists, contact your IBM representative.
0	RXSUBCOM_OK	The subcommand has been executed successfully.
0	RXFUNC_OK	The call to the API completed successfully.
0	RXMACRO_OK	The call to the API completed successfully.
1	RXMACRO_NO_STORAGE	There was not enough memory to complete the requested function.
2	RXMACRO_NOT_FOUND	The requested function was not found in the macrospace.
3	RXMACRO_EXTENSION_REQUIRED	An extension is required for the file name passed to the function.
4	RXMACRO_ALREADY_EXISTS	Duplicate functions cannot be loaded from the requested file.
5	RXMACRO_FILE_ERROR	An error occurred accessing the requested file.
6	RXMACRO_SIGNATURE_ERROR	The file specified as containing macrospace function images is not valid.
7	RXMACRO_SOURCE_NOT_FOUND	The requested file was not found.
8	RXMACRO_INVALID_POSITION	An invalid search-order position request flag was passed to the function.
10	RXSUBCOM_DUP	A duplicate environment name has been registered successfully; there is either: • An .EXE environment by the same name registered in another process; or • A DLL environment by the same name registered in another DLL; to address this subcommand, its library name must be specified.
10	RXFUNC_DEFINED	The requested function is already defined in the table.
20	RXSUBCOM_MAXREG	Cannot register anymore blocks.
20	RXFUNC_NOMEM	There is not enough memory to add a new function to the table.

Number	Name	Explanation
30	RXSUBCOM_NOTREG	This indicates either: • Registration was unsuccessful due to duplicate environment and dynalink names (RxSubcom Register); or • The subroutine environment is not registered (other RxSubcom functions).
30	RXFUNC_NOTREG	The requested function is not registered in the table.
40	RXSUBCOM_NOCANDROP	The subroutine has been registered as "not droppable."
40	RXFUNC_MODNOTFND	The dynamic link library module could not be found.
50	RXSUBCOM_LOADERR	An error has occurred while loading a routine into memory; most commonly this is due to a missing file.
50	RXFUNC_ENTNOTFND	The function entry point could not be found.
127	RXSUBCOM_NOPROC	The RxSubcom routine was not found; check for a missing file.

Alphabetic Order

Name	Number	Explanation
RXFUNC_DEFINED	10	The requested function is already defined in the table.
RXFUNC_ENTNOTFND	50	The function entry point could not be found.
RXFUNC_NOMEM	20	There is not enough memory to add a new function to the table.
RXFUNC_NOTREG	30	The requested function is not registered in the table.
RXFUNC_MODNOTFND	40	The dynamic link library module could not be found.
RXFUNC_OK	0	The call to the API completed successfully.
RXMACRO_ALREADY_EXISTS	4	Duplicate functions cannot be loaded from the requested file.
RXMACRO_EXTENSION_REQUIRED	3	An extension is required for the file name passed to the function.
RXMACRO_FILE_ERROR	5	An error occurred accessing the requested file.
RXMACRO_INVALID_POSITION	8	An invalid search-order position request flag was passed to the function.
RXMACRO_NO_STORAGE	1	There was not enough memory to complete the requested function.
RXMACRO_NOT_FOUND	2	The requested function was not found in the macrospace.
RXMACRO_OK	0	The call to the API completed successfully.
RXMACRO_SIGNATURE_ERROR	6	The file specified as containing macrospace function images is not valid.
RXMACRO_SOURCE_NOT_FOUND	7	The requested file was not found.
RXSUBCOM_ISREG	0x01	The subcommand environment has been registered.

Name	Number	Explanation
RXSUBCOM_ERROR	0x01	An error in execution has occurred; the interpreter raises an ERROR condition.
RXSUBCOM_FAILURE	0x02	A failure in execution has occurred; the interpreter raises a FAILURE condition.
RXSUBCOM_NOEMEM	1002	There is insufficient memory available to complete this request.
RXSUBCOM_BADTYPE	1003	Incorrect registration type; an internal failure may have occurred. Retry the operation; if this error persists, contact your IBM representative.
RXSUBCOM_OK	0	The subcommand has been executed successfully.
RXSUBCOM_DUP	10	A duplicate environment name has been registered successfully; there is either: • An .EXE environment by the same name registered in another process, or • A DLL environment by the same name registered in another DLL; to address this subcommand, its library name must be specified.
RXSUBCOM_MAXREG	20	Cannot register anymore blocks.
RXSUBCOM_NOTREG	30	This indicates either: • Registration was unsuccessful due to duplicate environment and dynalink names (RxSubcom Register); or • The subroutine environment is not registered (other RxSubcom functions).
RXSUBCOM_NOCANDROP	40	The subroutine has been registered as "not droppable."
RXSUBCOM_LOADERR	50	An error has occurred while loading a routine into memory; most commonly this is due to a missing file.
RXSUBCOM_NOPROC	127	The RxSubcom routine was not found; check for a missing file.

Index

A

API return codes
  invoking REXX interpreter 2-3
  shared variable pool 6-1
application programming interfaces
  external function interface
    RxFunctionCall 4-4
    RxFunctionDeregister 4-6
    RxFunctionQuery 4-7
    RxFunctionRegister 4-8
  interface data structures
    RXSTRING 1-2
    RXSYSEXIT 1-3
    SCBLOCK 3-10
  invoking REXX interpreter
    REXXSAA 2-1
  shared variable pool
    RxVar 6-3
  subcommand handling
    RxSubcomDrop 3-4
    RxSubcomExecute 3-5
    RxSubcomLoad 3-7
    RxSubcomQuery 3-8
    RxSubcomRegister 3-10
  system exits 7-1
    RxExitDrop 7-2
    RxExitQuery 7-2
    RxExitRegister 7-2
asynchronous request interface 7-19
  RxHaltSet 7-20
  RxTraceReset 7-22
  RxTraceSet 7-21

D

DosAllocHuge 1-3, 2-2, 4-2, 7-14
DosAllocSeg 2-2, 4-2, 7-14
DosFreeSeg 2-3, 3-5, 6-6
DosGetProcAddr 2-3, 3-11, 6-3
DosGiveSeg 4-2
DosLoadModule 2-3, 3-1, 3-2

E

error descriptions A-1
external functions
  calling 4-2
  function packages 4-1
  programming interface 4-1
  search order 5-1

F

function descriptions, conventions used iii
function libraries (macrospace) 5-1
  loading from a file 5-6
  saving to a file 5-10

I

in-storage procedures 2-2
invoking interpreter 2-1

M

macrospace interface
  function registration 5-1, 5-3, 5-6
  libraries
    See function libraries (macrospace)
  programming interface
    RxMacroChange 5-3
    RxMacroDrop 5-4
    RxMacroErase 5-5
    RxMacroLoad 5-6
    RxMacroQuery 5-8
    RxMacroReOrder 5-9
    RxMacroSave 5-10
  search order 5-1
    position flags 5-9

R

RC (return code)
  set by subcommands 3-2
returning result values 4-2
REXXSAA call 2-2
RXCMD exit service 7-7
RXCMDHST exit subfunction 7-7
RXENDLST 7-2
RxExitDrop system exit call 7-2
RxExitQuery system exit call 7-2
RxExitRegister subcommand call 7-2
RxExitRegister system exit call 7-2
RXFNC exit service 7-5
RXFNCCAL exit subfunction 7-5
RxHaltSet asynchronous request interface 7-20
RXHLT exit service 7-15
RXHLTCLR exit subfunction 7-15
RXHLTTST exit subfunction 7-15
RXMSQ exit service 7-9
RXMSQNAM exit subfunction 7-11
RXMSQPLL exit subfunction 7-9
RXMSQPSH exit subfunction 7-9
RXMSQSIZ exit subfunction 7-10
RXSIO exit service 7-12
RXSIODTR exit subfunction 7-14
RXSIOSAY exit subfunction 7-12
RXSIOTRC exit subfunction 7-12
RXSIOTRD exit subfunction 7-13
RXSTRING structure 1-2
RxSubcomDrop subcommand call 3-4
RxSubcomExecute subcommand call 3-5
RxSubcomLoad subcommand call 3-7
RxSubcomQuery subcommand call 3-8
RxSubcomRegister subcommand call 3-10
RXSYSEXIT 7-2
RXSYSEXIT structure 1-3
RXTER exit service 7-19
RXTEREXT exit subfunction 7-19
RxTraceSet asynchronous request interface 7-21

RXTRC exit service 7-17, 7-18
RXTRCTST exit subfunction 7-17
RxVar 6-3, 7-2

S

sample programs iii
SCBLOCK structure 3-10
search order
  macrospace functions 5-1
  subcommand handlers 3-2
special variables
  RC 3-2, 4-2
  Result 4-2
subcommands
  data definitions 3-10
  profile 3-2
  programming interface 3-1
    See also application programming interfaces, subcommand handling
  writing subcommand handlers 3-1
system exit handlers 7-1
  enabling system exits 7-2
  entry conditions 7-1
  exit conditions 7-1
  identifying handlers 7-2
system exits
  asynchronous request interface 7-19
    RxHaltSet 7-20
    RxTraceReset 7-22
    RxTraceSet 7-21
  exit 7-4
  exit characteristics 7-4
  exits 7-2
  External function exit 7-5
  External HALT exit 7-15
  external trace exit 7-17
  Host command exit 7-7
  initialization exit 7-18
  interface calls 7-2
  programming interface 7-1
  Queue exit 7-9
  RXCMD exit 7-7
  RXFNC exit 7-5
    RXCMDHST exit 7-7
    RXFNCCAL exit 7-5
    RXINIEXT exit 7-18
    RXTEREXT exit 7-19
    RXTRCTST exit 7-17
  RXHLT exit 7-15
    RXHLTCLR exit 7-15
    RXHLTTST exit 7-15
  RXINI exit 7-18
  RXMSQ exit 7-9
    RXMSQNAM exit 7-11
    RXMSQPLL exit 7-9
    RXMSQPSH exit 7-9
    RXMSQSIZ exit 7-10
  RXSIO exit 7-12
    RXSIODTR exit 7-14
    RXSIOSAY exit 7-12
    RXSIOTRC exit 7-12
    RXSIOTRD exit 7-13
  RXTER exit 7-19
  RXTRC exit 7-17
  Session I/O exit 7-12
  system exit handlers 7-1
system exits *(continued)*
  termination exit 7-19

T

tokenization of REXX procedures 2-2, 2-3

V

variable pool programming interface
  direct interface 6-4
  dropping variables 6-5
  fetching variables 6-4
    private information 6-5
  function codes 6-1, 6-4
  limitations 6-6
  return codes 6-1
  RxVar API function 6-3
  setting return values 6-6
  setting variables 6-4
  shared variable request block 6-1
  symbolic interface 6-4

IBM United Kingdom
International Products Limited
PO Box 41, North Harbour
Portsmouth, PO6 3AU
England

Printed in Denmark by Rosendahl · Esbjerg